UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMATICA
Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Microplanificación de asignación tardía y almacenamiento temporal distribuidos para flujos de trabajo intensivos en datos

Distributed Late-binding Micro-scheduling and Data Caching for Data-Intensive Workflows

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Antonio Delgado Peris

Directores

José María Hernández Calama
Eduardo Huedo Cuesta

Madrid, 2015

Microplanificación de asignación tardía y almacenamiento temporal distribuidos para flujos de trabajo intensivos en datos

Distributed Late-binding Micro-scheduling and Data Caching for Data-Intensive Workflows

TESIS DOCTORAL

Antonio Delgado Peris

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Documento maquetado con TEXIS v.1.0+.

Microplanificación de asignación tardía y almacenamiento temporal distribuidos para flujos de trabajo intensivos en datos

---

Distributed Late-binding Micro-scheduling and Data Caching for Data-Intensive Workflows

Memoria que presenta para optar al título de Doctor en Informática

Antonio Delgado Peris

Dirigida por los Doctores

José María Hernández Calama y Eduardo Huedo Cuesta

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática

Universidad Complutense de Madrid

Junio 2015

Para la realización de este trabajo se utilizaron recursos facilitados por el Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas, al cual expresamos nuestro agradecimiento.

Asímismo, queremos agradecer la financiación recibida desde la Secretaría de Estado de Investigación, Desarrollo e Innovación a través de los proyectos *FPA2010-21638-C02-02* y *FPA2013-47804-C2-1-R*.

It might easily argued that human beings have no right to say that this or that is impermissible; that something that is called supernatural receives its name by arbitrary definition out of knowledge that is finite and incomplete. [...]
Quite right, but consider this:
When we lead from ignorance, we can come to no conclusions. When we say, Anything can happen, and anything can be, because we know so little that we have no right to say This is or This isn't, then all reasoning comes to a halt right there. We can eliminate nothing; we can assert nothing. All we can do is put words and thoughts together on the basis of intuition or faith or revelation and, unfortunately, no two people seem to share the same intuition or faith or revelation.
What we must do is set rules and place limits, however arbitrary these may seem to be. We then discover what we can say within these rules and limits.

Isaac Asimov

A Paula y a Sandra, a Vero, a papá y mamá.

Agradecimientos

Tout a été dit, mais comme personne
n'écoute, il faut toujours répéter.

André Gide

Es tan común encontrar una sección de agradecimientos al principio de una tesis doctoral que uno puede acabar pensando que se hace más por costumbre, por protocolo, que por ningún otro motivo. Nada más lejos de la realidad. La realización de una tesis es un proceso tan largo, cuesta tanto llegar hasta el final (por unas cosas o por otras), que uno realmente tiene *mucho* que agradecer a *mucha gente* (y unas inmensas ganas de poder hacerlo por fin). En mi caso, además, es la ocasión para corresponder a los recientes doctores que, a lo largo de los últimos años, me han ido incluyendo en sus reconocimientos (hecho que supone un motivo de profunda honra para mí). Por si todo eso fuera poco, cualquier excusa es buena para poder dar las gracias a tantas personas que realmente se lo merecen. Así que... allá voy.

Yo empecé con esto de la tesis hace ya más de seis años; dos más, incluso, si contamos el correspondiente máster. Y el camino no siempre ha sido fácil. Compaginar el trabajo con el doctorado ha exigido muchas horas delante de la pantalla; tiempo robado al descanso, al ocio, a amigos y familia. Y aunque durante todo el recorrido he disfrutado mucho y he aprendido aún más, no han faltado los trances complicados, momentos en los que uno se pregunta si realmente conseguirá llegar a la meta alguna vez.

Pero parece que por fin está escrita.

Y desde luego no habría podido llegar hasta aquí sin la guía y los consejos de mis directores, Eduardo y Chema, para los que solo puedo tener buenas palabras. Chema, con quien comparto, no solo la tesis, sino la rutina de CMS y muchas charlas de pasillo, y de cuyas palabras y sus formas de hacer llevo aprendiendo tanto tiempo ya. Eduardo, sin cuya perspectiva y cuya dirección a través de los casi insondables caminos de la publicación científica y las tramitaciones académicas nunca habría concluido esta aventura con éxito. Por supuesto, también he de agradecer la ayuda de Khawar y los demás colaboradores de las primeras etapas del proyecto del *Task Queue*.

También imprescindible ha sido el apoyo de mis compañeros de computación (*rara avis* en nuestro departamento). En primer lugar, Javier, quien, no solo me ha ayudado infinitas veces, cada día, tanto en lo referente a la tesis como fuera de ella, sino que se ha apresurado a liberarme

de trabajo, echándoselo sobre sus espaldas, en las fases en las que las exigencias del doctorado minaban mi dedicación a otras tareas. Desde luego, de no ser por él, no estarías leyendo estas líneas. También le estoy agradecido a Nica por, no solo permitirme, sino animarme a embarcarme en esta empresa, y por compartir su experiencia. A Miguel, por sus innumerables y bien aprovechados consejos. A Juanjo, por echar una mano siempre que ha hecho falta. Y a todos ellos (y a Raúl y Jaime), por su talante y por hacer más fácil y más ameno el trabajo diario.

A partir de ahí, la lista de colegas *ciemateros* a los que debo gratitud es tan larga que no voy a intentar mencionarlos a todos. En este tiempo he convivido con un gran número de compañeros, muchos estudiantes, y un buen puñado de ellos hoy ya doctores. Todos han contribuido a hacer mi camino indescriptiblemente más agradable. Los incontables cafés y comidas, el deporte y la montaña, los cines, las cenas, los premios de navidad, los momentos de risas y las conversaciones estimulantes. No tengo dudas de que la experiencia compartida con ellos es lo más valioso de todo este periplo. Así que, en fin, muchas gracias a todos, allá donde estéis, ya sea Ginebra o Hamburgo, Río de Janeiro o Chicago, Quito o Zúrich. Y, por supuesto, gracias a los que resisten todavía por Madrid, dentro o fuera del CIEMAT (incluido, sí, el otro *perenne*, con quien tantas andanzas he vivido ya).

Fuera del entorno laboral, me gustaría recordar a la gente que me ha sufrido todos estos años, incluso desde antes de comenzar la tesis, y que hacen la vida más dulce. A los *ibmeros* y los (ex-)*cernícolas*, a mis compis de piso (de Pablo a Shu, pasando por Mikel y Bea, y, por supuesto, Javi) y a la gente de Valencia a la que veo mucho menos de lo que querría y que algo habrán tenido que ver en todo esto (Boro, Cuñi, Josemi, Óscar, Juanfran, Mario, MC... y todos los demás). A ellos y a todos los que olvido. Pero no quiero cerrar el párrafo sin mencionar a mi profe de mates de octavo (por descubrirme la programación), a Vicente (por su *excel*), a Andreu (por mucho más que por ser un pesado), a Wisconsin (por no ser Groenlandia) y a Michael (porque me apetece).

Y ya solo me restan los más importantes. Mi familia; en Málaga, Valencia y Madrid. A todos ellos sin excepción, pero dejadme nombrar a mis abuelas, Carmen y Vicenta, que ojalá estuvieran, a mi tía Paquita, a la que adoro, a Sandra y a Paula, las mejores hermanas que nadie pueda soñar y, por supuesto, a papá, por todo. A ellos y a Vero, que fue la primera en leer estas páginas (pobrecita) y que ha soportado mis dudas y lamentos más que nadie, pero que, a pesar de todo, sigue caminando conmigo, mano sobre mano, iluminando el trayecto a cada paso. Por último, a mamá, la persona que más culpa tiene de que haya llegado hasta aquí, no solo por el detalle de haberme engendrado sino por ayudarme y estimularme desde siempre, por darme apoyo y cariño sin mesura, por su sabiduría, su alegría y su pasión.

Muchas gracias.

Abstract

Today's world is flooded with vast amounts of digital information coming from innumerable sources. Moreover, it seems clear that this trend will only intensify in the future. Industry, society and—remarkably—science are not indifferent to this fact. On the contrary, they are struggling to get the most out of this data, which means that they need to capture, transfer, store and process it in a timely and efficient manner, using a wide range of computational resources. And this task is not always simple. A very representative example of the challenges posed by the management and processing of large quantities of data is that of the *Large Hadron Collider* experiments, which handle tens of petabytes of physics information every year. Based on the experience of one of these collaborations, we have studied the main issues involved in the management of huge volumes of data and in the completion of sizeable workflows that consume it.

In this context, we have developed a general-purpose architecture for the scheduling and execution of workflows with heavy data requirements: the *Task Queue*. This new system builds on the late-binding overlay model, which has helped experiments to successfully overcome the problems associated to the heterogeneity and complexity of large computational grids. Our proposal introduces several enhancements to the existing systems. The execution agents of the Task Queue architecture share a Distributed Hash Table (*DHT*) and perform job matching and assignment cooperatively. In this way, scalability problems of centralized matching algorithms are avoided and workflow execution times are improved. Scalability makes fine-grained *micro-scheduling* possible and enables new functionalities, like the implementation of a distributed data cache on the execution nodes and the integration of data location information in the scheduling decisions. This improves the efficiency of data processing and helps alleviate the commonly congested grid storage services. In addition, our system is more resilient to problems in the central server and behaves better in scenarios with demanding data access patterns or with no local storage service available, as an extensive set of assessment tests has proven.

Since our distributed task scheduling procedure requires the use of broadcast messages, we have also performed an exhaustive study of the possible

approaches to implement this operation on top of the *Kademlia DHT*, which was already used for the shared data cache. Kademlia provided individual node routing but no broadcast primitive. Our work exposes the particularities of this system, notably its XOR-based distance metrics, and analytically studies which broadcasting techniques can be applied to it. A model that estimates node coverage as a function of the probability that individual messages reach their destination has also been developed. For validation, the algorithms have been implemented and comprehensively evaluated. Moreover, several techniques are proposed to enhance the bare protocols when adverse circumstances such as churn and failure rate conditions are present. These include redundancy, resubmissions or flooding, and also combinations of those. An analysis of the strengths and weaknesses of algorithms and additional techniques is presented.

Resumen

El mundo de hoy en día se encuentra inundado por ingentes cantidades de información digital procedente de muy diversas fuentes. Todo apunta, además, a que esta tendencia se agudizará en el futuro. Ni la industria, ni la sociedad en general, ni, muy particularmente, la ciencia, permanecen indiferentes ante este hecho. Al contrario, se esfuerzan por obtener el máximo provecho de esta información, lo que significa que deben capturarla, transferirla, almacenarla y procesarla puntual y eficientemente, utilizando una amplia gama de recursos computacionales. Pero esta tarea no es siempre sencilla. Un ejemplo representativo de los desafíos que suponen el manejo y procesamiento de grandes cantidades de datos es el de los experimentos de física de partículas del *Large Hadron Collider* (*LHC*), en Ginebra, que cada año deben gestionar decenas de *petabytes* de información. Basándonos en la experiencia de una de estas colaboraciones, hemos estudiado los principales problemas relativos a la gestión de volúmenes de datos masivos y a la ejecución de vastos flujos de trabajo que necesitan consumirlos.

En este contexto, hemos desarrollado una arquitectura de propósito general para la planificación y ejecución de flujos de trabajo con importantes requisitos de datos, que hemos llamado *Task Queue*. Este nuevo sistema aprovecha el modelo de asignación tardía basado en agentes que ha ayudado a los experimentos del LHC a superar los problemas asociados con la heterogeneidad y la complejidad de las grandes infraestructuras *grid* de computación. Nuestra propuesta presenta varias mejoras con respecto a los sistemas existentes. Los agentes de ejecución de la arquitectura *Task Queue* comparten una tabla *hash* distribuida (*Distributed Hash Table*, *DHT*) y realizan la asignación de tareas de una manera cooperativa. De esta forma, se evitan los problemas de escalabilidad de los algoritmos centralizados de asignación y se mejoran los tiempos de ejecución. Esta escalabilidad nos permite realizar una *microplanificación* de grano fino lo cual posibilita nuevas funcionalidades, como la implementación de una *cache* distribuida en los nodos de ejecución y el uso de la información de ubicación de los datos en las decisiones de asignación de tareas. Esto mejora la eficiencia del procesado de datos y ayuda a aliviar los habitualmente congestionados servicios de almacenamiento del *grid*. Además, nuestro sistema es más robusto frente

a problemas en la interacción con la cola central de tareas y ofrece mejor comportamiento en situaciones con patrones de acceso a datos exigentes o en ausencia de servicios de almacenamiento locales. Todo esto ha sido demostrado en una amplia serie de pruebas de evaluación.

Dado que nuestro procedimiento de planificación de tareas distribuido requiere el uso de mensajes de *broadcast*, también hemos realizado un profundo estudio de las posibles aproximaciones a la implementación de esta operación sobre el *DHT Kademlia*, el cual es utilizado para la *cache* de datos compartida. Kademlia ofrece enrutamiento a nodos individuales pero no incluye ninguna primitiva de *broadcast*. Nuestro trabajo expone las peculiaridades de este sistema, particularmente su métrica basada en la operación *XOR*, y estudia analíticamente qué técnicas de *broadcast* pueden ser usadas con él. También se ha desarrollado un modelo que estima la cobertura de nodos en función de la probabilidad que cada mensaje individual alcance su destino correctamente. Como validación, los algoritmos se han implementado y se han evaluado exhaustivamente. Además, proponemos varias técnicas para mejorar los protocolos en situaciones adversas, por ejemplo cuando el sistema presenta una alta rotación de nodos o la tasa de error en las entregas no es despreciable. Esta técnicas incluyen redundancia, reenvío e inundación (*flooding*), así como combinaciones de las mismas. Presentamos un análisis de las fortalezas y debilidades de los diferentes algoritmos y las mencionadas técnicas complementarias.

Publications and Personal Contribution

The work on this thesis has led to the publication of several articles.

Material discussed in Chapter 6 was published in the following articles:

- A. Afaq, B. Bockelman *et al. The Evolution of the Data Location Service in CMS.* In 3rd Iberian Grid Infrastructure Conference Proceedings, pp. 189–200. Netbiblo, 2010.
- A. Delgado Peris, J. Hernández and E. Huedo *Data Location-aware Job Scheduling in the Grid. Application to the GridWay Metascheduler.* In Journal of Physics: Conference Series, vol. 219, p. 062043. IOP Publishing, 2010.

Most of the contents of Chapter 7 were published in:

- K. Hasham, A. Delgado Peris *et al. CMS Workflow Execution Using Intelligent Job Scheduling and Data Access Strategies.* IEEE Transactions on Nuclear Science, vol. 58(3), pp. 1221–1232, IEEE, 2011.

Large parts of Chapter 9 were included in the following papers:

- A. Delgado Peris, J. M. Hernández and E. Huedo. *Evaluation of the Broadcast Operation in Kademlia.* In IEEE 14th Intl. Conf. on High Performance Computing and Communication & IEEE 9th Intl. Conf. on Embedded Software and Systems (HPCC-ICESS), pp. 756–763. IEEE Computer Society, 2012.
- A. Delgado Peris, J. M. Hernández and E. Huedo. *Evaluation of Alternatives for the Broadcast Operation in Kademlia under Churn.* Peer-to-Peer Networking and Applications. Springer, 2015. DOI: `http://dx.doi.org/10.1007/s12083-015-0338-y`

Chapter 10 was partly built using the material originally appeared in:

- A. DELGADO PERIS, J. M. HERNÁNDEZ and E. HUEDO. *Distributed Scheduling and Data Sharing in Late-binding Overlays.* In High Performance Computing Simulation (HPCS), 2014 Intl. Conf. on, pp. 129–136, July 2014.

In addition, a new article has been submitted and it is currently being revised. This paper includes work discussed in Chapters 5, 8 and 10:

- A. DELGADO PERIS, J. M. HERNÁNDEZ and E. HUEDO. *Distributed Late-binding Scheduling and Cooperative Data Caching.* Submitted to *Journal of Grid Computing.*

Personal contribution

During the time I have been working on this thesis, I have been certainly assisted with the advise and support of many different people and, most especially, my supervisors and the other co-authors of the previously listed articles. However, I am the main responsible for all the material presented throughout the thesis and in the articles listed above (including the writing of the articles themselves).

The only partial exception to this statement is the work discussed in the article *CMS Workflow Execution Using Intelligent Job Scheduling and Data Access Strategies* (and the related Chapter 7), which was produced in close cooperation with Kawhar Hasham. This was the first phase of the design and implementation of the Task Queue system (when it still used a centralized matching architecture).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

Everyday, more and more people spend hours in social networking sites, upload pictures to some sharing website, watch videos online or play video games over the network. The number of Internet-capable portable devices rises continuously and so does the number of pixels per inch of digital cameras. Advances in digital sensors, communications, computation and storage make it possible to produce, transfer and store ever increasing volumes of data. Also in the business domain, we can find countless examples of activities generating large amounts of data: online shopping, recording of transactions at point-of-sale terminals or analysis of activity patterns on cloud-based services.

This pattern of data superabundance is also found in the scientific domain. Astronomers process telescope data to produce sky mosaics and examine the structure of the galaxies. Bioinformaticians and medical researchers study large samples of magnetic resonance images, genome sequences and protein structure databases in order to reach a better understanding of the causes of diseases and to create more effective means of diagnosis and treatment. In meteorology and earth science, sensor measurements are key to understand the complex interactions of the different agents and to make more accurate predictions [1, 2].

Maybe the most representative example of this trend is the field of high energy physics and, in particular, the LHC (*Large Hadron Collider*). The collisions occurring at the accelerator, at Geneva, generate several petabytes of data annually. This data must be timely stored and replicated to computing centres around the world for processing and further archival. The processed products must be carefully analyzed to extract physics results on the laws that govern nature at the microscopic level. The infrastructure used to satisfy the storage and processing requirements of the LHC is called WLCG (*Woldwide LHC Computing Grid*), which is currently the world's

largest computational grid.

Our work was originally motivated by the observation of the congestion problems suffered by one of the LHC experiments, CMS (*Compact Muon Solenoid*), when running large workflows that consumed extensive amounts of data. This resulted in longer workload completion times and inefficient use of storage and processing resources. We initiated a research to better understand the challenges that the execution of data-intensive workflows in WLCG entails and produce a more effective architecture to deal with them.

1.2. Research Objectives

The main objective of this thesis is the study of the scheduling and execution of large data-intensive workflows on distributed computing resources and the development of a new architecture to improve their performance (both regarding completion times and data access). In addition, since broadcasting was required for the new system, an analysis of this operation on top of the Kademlia DHT (*Distributed Hash Table*) also became a major target for our work.

More specifically, the main research objectives addressed by this thesis are the following:

- Study the problem of data-intensive scheduling in distributed computing resources.
- Since they are the current state of the art for workload management in WLCG, analyze the operation of late-binding overlay architectures, their advantages and the challenges they face.
- Propose a way to improve data access efficiency and protect massive storage systems: a distributed data cache.
- Propose an advanced scheduling system that enables the use of data location information in the scheduling decisions: micro-scheduling.
- Develop an effective and scalable procedure to perform these scheduling: DHT-based cooperative algorithm.
- Study existing and new algorithms to perform the broadcast operation in the Kademlia DHT.
- Evaluate the risks associated to churn and loss of messages when propagating broadcasts in Kademlia. Propose and assess techniques to overcome these problems.
- Implement a broadcast-capable network that satisfies the needs of our architecture.

- Implement the complete architecture with distributed data cache and task scheduling and carry out a comprehensive evaluation of its functionality and performance in different conditions.

1.3. Organization of the Document

This document is divided in four main parts and one appendices section.

Part I introduces the issues dealt with by this thesis and reviews related work. Chapter 2 sets the context of data abundance that we are experiencing in recent times and presents the main technologies and infrastructures used for massive distributed computing today. Chapter 3 examines workload and data management in WLCG, providing a detailed comparison of traditional job scheduling and the new late-binding paradigms and reviewing academic and commercial approaches to the execution of data-intensive workflows. Chapter 4 discusses DHT systems, particularly Kademlia, and mentions existing proposals to add broadcast capabilities to theses systems.

Part II delves into the problem of executing data-intensive workloads. Chapter 5 summarizes the specific problems of data access and scheduling overhead and outlines our proposals for data caching and micro-scheduling. Chapter 6 shows the importance of data-location awareness when scheduling workloads with heavy data requirements and discusses our early work on this topic. Chapter 7 presents and evaluates the Task Queue architecture, a new late-binding overlay system enhanced with data caching and micro-scheduling.

Part III discusses the introduction of DHT technologies to achieve a more effective distributed data cache and a scalable cooperative task scheduling and assignment procedure. Chapter 8 reviews the weaknesses shown by the original Task Queue architecture and motivates the need for a new distributed approach. Chapter 9 provides a deep study of the broadcast operation in the Kademlia DHT, which is required for our distributed matching algorithm. Chapter 10 presents our final architecture and shows a complete series of tests that validate its functionality and performance.

Finally, Part IV discusses the conclusions of the thesis and Appendix A provides some additional information on the details of the internal architecture and implementation of some systems discussed throughout the document.

Part I

Large-scale Distributed Data-intensive Computing

Chapter 2

Large-scale Computing

Modern science needs massive computing. Today's research produces ever increasing volumes of data, which require higher and higher processing, transferring and storing capabilities. It has been stated that computation has become an established third branch of science, alongside theory and experiment, and this use of computer technology in scientific investigation has been called *e-science* [3].

In order to fulfill their enormous computing requirements, scientists use a wide range of distributed computing infrastructures, from computing clusters and GPU (*Graphics Processing Unit*) processors to grids, clouds and volunteer computing systems. Moreover, scientists often need to exploit remote databases and repositories. Frequently, research collaborations incorporate members from many different institutions from all over the world. The distributed structure of the community itself encourages the integration of scattered computational resources (those provided by the participating centres) and makes it necessary for the key services to be accessible by remote collaborators, which implies that the proper authentication and authorization mechanisms must be in place.

Large-scale grids for scientific computing, discussed in Section 2.1, probably constitute the most dramatic example of a complex distributed computing system. They comprise heterogeneous resource centres (sites) spread under multiple administrative domains and interconnected by complex network infrastructures, where high latencies are not uncommon. In such systems, applications must deal with multiple interfaces, highly heterogeneous execution environments, dynamic number of available resources and routine component failures and maintenance downtimes.

The cloud came as a commercial alternative to (and an evolution of) the grid. With a marked accent on ease of use, the introduction of powerful virtualization techniques and, perhaps, somewhat less ambitious goals, *cloud computing* has been more successful than the grid (by which it was probably inspired) in attracting companies and consumers attention but it

is also being used for scientific research. We will discuss cloud computing in Section 2.2.

More generally, data is being generated by everything around us—sensors, cameras, smartphones, tablets, computers—at all times and the world is now trying to extract knowledge out of it—identifying patterns and trends, evaluating risks, understanding opportunities and threats. We live in the era of *big data* and the task of managing and analyzing this information requires large resource pools and demands new computing paradigms. Section 2.3 introduces big data and its relation to data-intensive computing research.

2.1. The Grid

2.1.1. The Grid Vision

The *grid*[1] vision was presented in 1999 [4]. Since that moment and for several years, grid computing was one of the most active fields of research, especially in academia. Countless research articles were published on the subject and numerous grid-related projects were initiated. To give just a couple of examples of the attention received by the domain, in 2003, the MIT Technology Review named it as one of the *10 technologies that would change our world* [5]; also, *Grid computing has been hailed as the next revolution after the Internet and the World Wide Web* [6].

During those years (and still today), the *grid* term was used in a variety of contexts and its meaning was not always consistent. In reality, the grid idea never referred to a concrete, well-defined technology, but rather, the vision of a new computing paradigm, encompassing concepts such as *metacomputing* and *utility computing*.

The first one refers to the integration of multiple computing resources for a particular application (going beyond individual supercomputers or even clusters). This goal derives from the observation that some projects cannot be fulfilled with local resources alone. While HPC (*High Performance Computing*) applications have very demanding needs with require special hardware (supercomputers), others, HTC (*High Throughput Computing*) applications, can be parallelized to utilize large amounts of commodity computers. Grid technologies should enable this sharing of computing resources from different sources.

The second concept—utility computing—was proposed as soon as 1966

[1] In the literature, the term *grid* is sometimes capitalized. The reasons for this are probably twofold: firstly, it is a way to distinguish it from the traditional uses of the word; secondly, because there was once the idea that at some point there would be a single *Grid*—much like there is an *Internet*—so it could be regarded as a proper noun. The same can be said of the *the cloud*. The reality, however, is that there is currently no integrated grid or cloud, but multiplicity of technologies and infrastructures. Therefore, we see no reason to capitalize such words and we will use the lowercase form throughout the text.

and suggests that computing resources be provisioned by external service providers and their consumption charged for specific usage (like gas or water supplies) [7]. Actually, the word *grid* was chosen as an analogy to the electric grid, in which consumers of computing power would not necessarily own their resources (like they do not own an electric generator) but acquire it from external providers using the data communication networks (like the electricity is received through power cables). In this model, consumers do not need to own the means required to satisfy their peak computing needs but can rent them from the external providers when desired. Moreover, these resources will typically be better managed, more secure and, thanks to economies of scale, might even be cheaper than those that they could buy by themselves.

Even if it is not easy to provide a concise, concrete and widely accepted definition of what grid computing is (and what is not), we believe that it is more practical to focus on the deployed grid infrastructures and the developed grid technologies. In general terms, we can say that a *grid infrastructure* is a dynamically changing set of computing resources distributed among different administrative domains and that *grid technologies* are those tools and techniques that enable users to select, access and aggregate those resources in a seamless and secure way, using standard, open, general-purpose protocols and interfaces. Consumers of these resources are individuals and institutions grouped in what has been called VO (*Virtual Organization*). Each VO acts according to well-defined rules stating which grid resources are shared and who is allowed to access them and under which conditions [8, 9, 10].

The last part of the definition above, *using standard, open, general-purpose protocols and interfaces*, tries to separate grid computing from proprietary solutions. Vigorous efforts were performed in this respect, but, while the initial grid reference implementation, the Globus project [11], achieved great success and became the *de facto* standard for grid protocols (at least for some time), subsequent attempts of several organizations to define open grid standards did not really achieve widespread adoption [12]. In the search of efficiency or usability, different communities (and projects) ended up developing different solutions, often incompatible or of application to their particular domain only.

The vision of a global interoperable grid where only defined authorization policies limit access to innumerable resources worldwide was never reached. Moreover, the grid technologies and concepts, generally regarded as immature or too complicated for the average need[2], did not really find great acceptance outside the academic circles. Nevertheless, as of now, the Globus project still provides grid services to researchers around the world and several

[2]This should not be surprising, since these technologies were originated and evolved to satisfy the needs of the highly specialized HTC community.

large grid infrastructures devote their vast computing resources to scientific activities [13]. The most prominent examples of the latter are EGI (*European Grid Infrastructure*) [14], OSG (*Open Science Grid*) [15] and WLCG (*Woldwide LHC Computing Grid*) [16], which we will cover in more depth in Section 2.1.3.

Today, the buzz surrounding grid computing has disappeared. However, a new paradigm, to certain degree inspired by the former, has come to replace it: *cloud computing*. We will discuss this in Section 2.2.

2.1.2. Grid Middleware

The definition of grid that we have presented is general enough to leave space for virtually any type of computing resource to be shared; e.g., processing power, storage capacity, datasets or remote harnessing of any kind of scientific instrumentation. But even for a single type of resource, several different implementations may exist, possibly offering incompatible interfaces; especially since resources usually belong to different administrative domains. Such situations make it often impossible for an application to directly access all available resources in a seamless way. To fix this, a single interface must be defined for each type of resources. Through this uniform interface, any of the implementations can be accessed. But this is not enough, resources must also be advertised, using a coherent description schema, so that consumers can discover the existing services and choose the one they prefer. Finally, authentication, authorization and usage policies for users and services must be enforced at all times.

The software that provides the means to describe, discover and use available resources in a secure way, thus providing the backbone to the grid, is usually called *grid middleware* [8]. In order to enable sharing of any kind of resource, the middleware should make use of standard, open, general-purpose protocols and interfaces. This is opposite to application-specific solutions, but in fact, the latter are sometimes more easily realizable or can be more efficient, so they are commonly preferred for real applications. In practice, a mixture of general-purpose and specific products may be found in existing grid infrastructures.

The *Globus Toolkit*, released by the Globus project, was the first middleware implementation and soon became the reference. Its architecture follows the hourglass model: a small set of core abstractions and protocols are at the neck of the hourglass, while many different supported resources are at the bottom and numerous applications can be built upon them (top of the hourglass). The services offered by the Globus Toolkit include, amongst others, resource monitoring and discovery services, resource allocation and management, a public key security infrastructure and secure and scalable file transfer services (the *GridFTP* protocol). Despite the success of the initial versions of the Globus Toolkit, subsequent releases using stateful web

services, following the OGSA (*Open Grid Services Architecture*) model [12], did not achieve the same popularity.

A variety of middleware breeds, mostly built upon Globus, were developed within different research projects and organizations. They were meant to extend functionalities of the Globus Toolkit (based on the needs of particular communities) or to improve its performance. A remarkable example is the software produced under the umbrella of different European research projects, with the purpose of serving the European and international research community. Such middleware has been released with several different names throughout the years. Its current designation is EMI (*European Middleware Initiative*) [17], used in EGI and the different European NGIs (*National Grid Initiatives*). One important characteristic of EMI and its ancestors is that, in contrast to the bare Globus tools, it devotes great attention to data storage and management issues. This is due to the importance of the high energy physics experiments, known to manage large amounts of data, within EMI.

Without dismissing traditional grid middleware, like EMI and Globus, the current trends within the most important science grid infrastructures, EGI and OSG, are both to incorporate mainstream protocol and technologies as much as possible (e.g., making use of standard NFS or HTTP protocols for data handling, instead of possible grid alternatives [18]) and to make use of any product whose performance is desirable, even if this does not really adhere to open grid standards (this may be the case of some uses of HTCondor [19] for job execution or of the xrootd [20] protocol for remote data access). This is possible in these research communities that, although not small, are well organized and, relatively speaking, not very heterogeneous.

2.1.3. The Worldwide LHC Computing Grid

The LHC (*Large Hadron Collider*), at CERN (*European Laboratory of Particle Physics*), is the world's largest and most powerful particle collider, and the largest and most complex experimental facility ever built [21]. Developed by a collaboration of over 10,000 scientists and engineers from over 100 countries, the LHC started full operation in 2010, after nearly 20 years of preparation and construction. The LHC lies in a circular tunnel of 27 kilometres beneath the Franco-Swiss border near Geneva, Switzerland. Four detectors are operated by four particle and high-energy physics experiments: ALICE (*A Large Ion Collider Experiment*), ATLAS (*A Toroidal LHC Apparatus*), CMS (*Compact Muon Solenoid*) and LHCb (*Large Hadron Collider beauty*). In July 2012, ATLAS and CMS announced the existence of a new particle, consistent with the theoretical prediction of the Higgs boson, by François Englert and Peter Higgs, who were awarded with the Nobel Prize in Physics in 2013.

The LHC physics experiments aim at detecting very low probability pro-

cesses hidden in an overwhelming background of particle collisions, generating very large samples of data, in the form of discrete events. Including the data generated by the necessary physics and detector simulations, the LHC computing system has produced tens of petabytes of new data each of the last few years. These data must be processed and analyzed, using computational jobs.

Years before the LHC was operational, it became evident that, with the available funding, CERN alone would not be able to satisfy all the computing and storage requirements that the activities of the LHC experiments required. The solution to this problem was found in the local computing facilities of the associated institutes participating in the LHC experiments. A distributed system that made use of all the available resources was proposed. Eventually, grid technologies were chosen to implement such system, since they seemed to tackle exactly the problem at hand. This distributed resource model matched well with the funding structure of the entities involved with the LHC and should make it easier for physicists at the different institutes to gain access to the necessary data.

The Worldwide LHC Computing Grid is a large distributed computing infrastructure—more than 400,000 CPU (*Central Processing Unit*) cores, 300 PB of disk and more than 200 PB of tape—devoted to store, deliver and analyze the data generated by the LHC, making the data available to all partners, regardless of their physical location [22]. The WLCG is operated by a global collaboration of more than 170 computing centres, in 40 countries. The WLCG is supported by many associated national and international grids, such as EGI (Europe-based) and OSG (USA-based), as well as many other regional grids. While both EGI and OSG provide computation resources to researchers from many different scientific disciplines, the high energy physics community is their most important *consumer* (in terms of resource needs) and the main driving force behind the projects. As a whole, the WLCG is the world's largest computing grid [16]. More than 2 million jobs are run every day and these figures are bound to increase once the LHC starts its next physics run in summer 2015, after several years of shutdown [18].

To satisfy the needs of the LHC experiments, the WLCG has faced the following challenges:

- Manage very large data volumes at very high data rates.
- Provide the necessary CPU and storage capacities for processing, simulation, analysis and data preservation.
- Provide long-term data archiving in a robust way.
- Allow several thousands of physicists to access the data from their different home institutes all over the world.

- Manage both centrally organized data processing and chaotic user analysis (dependent on the evolving needs of the numerous analysis groups and individual physicists).

2.1.3.1. WLCG Architecture

In WLCG, each experiment is represented by a VO. Each resource centre (*site*) may decide to support one or more of these VOs. The sites are organized in *tiers*, according to their relative size (in terms of resources) and the functions they accomplish. Even if the duties of each tier vary from one experiment to the other, we can say that, in general terms, the centres are classified as follows:

- *Tier-0* (CERN): Receives the raw data from the detector, caches it on disk, archives it to tape and distributes a second copy among the Tier-1 sites. Calibration and alignment are also run at Tier-0 and a small sample of raw data is promptly processed for quality checks. In addition, a first-pass processing of the raw data is performed too (and the results archived and distributed to Tier-1 centres).

- *Tier-1*: Usually equipped with mass storage systems and archival facilities (tape), Tier-1 centres are responsible for reprocessing the raw data once adequate calibrations are available, as well as for distributing it to the collaboration. They also host simulated data produced at the Tier-2 sites.

- *Tier-2*: They host a share of the processed datasets and provide analysis facilities, as well as resources for producing certain amount of simulated data.

- *Tier-3*: Opportunistic resources, with little or no formal commitments (pledges) to LHC experiments, and mostly offering CPU only, with no permanent storage available.

In regard to the middleware, software products from several different sources are used in WLCG. This includes, among others, OSG, EMI (which was the middleware provider for EGI and which, in turn, integrated components from other projects, such as Globus or NorduGrid) and software produced by CERN and the LHC experiments themselves. Figure 2.1 shows the most important services in the WLCG, which are also described as follows.

- **Storage services**. A SE (*Storage Element*) provides storage services at a given site. Its interfaces are based on the SRM (*Storage Resource Manager*), which provides the same management interface, regardless

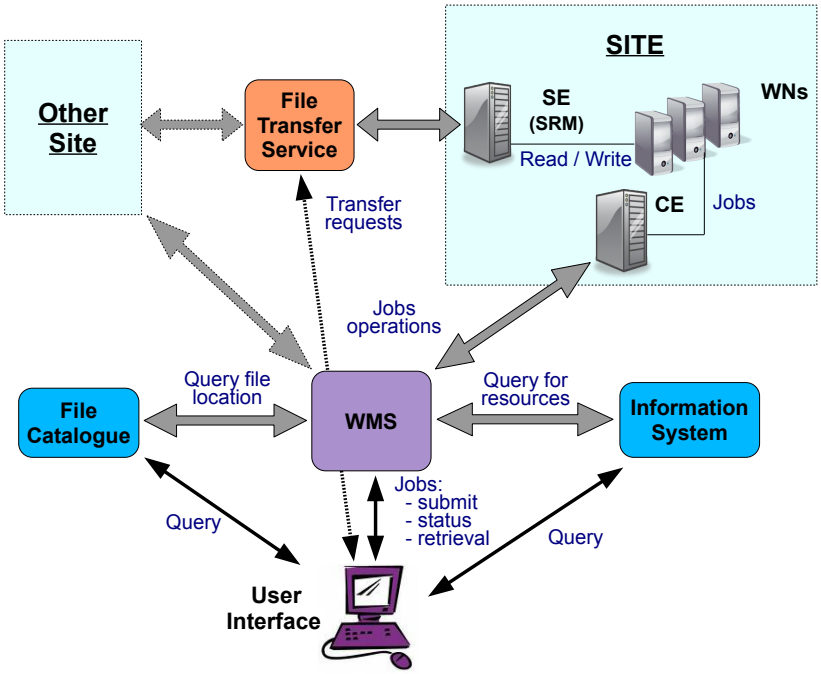


Fig. 2.1: Service architecture in the Worldwide LHC Computing Grid.

of the underlying storage technology and the available data access protocols [23]. However, all WLCG SEs support the GridFTP protocol for data transferring.

- **File transfer services**. They manage data transfers between SEs. They provide error recovery, retry on failure and a mechanism to share bandwidth between VOs.

- **Computing services**. The CE (*Computing Element*) is the interface to the computing facilities at a site (most often, a batch system in front of a computing cluster). It allows remote authorized clients to submit jobs, perform status queries and retrieve output files. Each of the hosts at the site where jobs are actually executed is called WN (*Worker Node*).

- **Workload management**. This function has been traditionally fulfilled by the WMS (*Workload Management System*), sometimes also called *resource broker* or *metascheduler*[3], which matched the available computing resources to the preference of clients, and submitted the

[3]Since the entity in charge of the scheduling of computing tasks in a traditional batch system is called *scheduler*, the service selecting the batch systems to submit jobs to, is often called *metascheduler*.

jobs to the appropriate CE [24]. However, as we will see, this functionality has been gradually moved into the workflow systems of the experiments.

- **File catalog and database services**. These services are usually run centrally at Tier-0 and Tier-1 sites. The file catalogs map grid data files to the SEs holding them while other databases contain alignment and calibration parameters of the detectors and other metadata.
- **Information service**. Aggregates and offers the information about existing resources, their configuration and status (e.g., number of running jobs and available slots at the CE).
- **Application software**. The experiment application software is regularly updated and must be made available at each site. A suite of standard utilities are also provided.

2.1.3.2. The Compact Muon Solenoid Experiment

The CMS experiment is an international scientific collaboration that investigates a wide range of elementary particle physics resulting from the operation of one of the four detectors at the LHC (the *CMS detector*) [25]. Their goals include the search for the Higgs boson, extra dimensions and particles that could produce dark matter. The CMS collaboration involves 4,300 physicists, engineers, technicians, students and support staff from 182 institutes in 42 countries (February, 2014).

Like the other LHC experiments, CMS has to deal with large amounts of data and requires huge computational power. In order to fulfill its duties, CMS makes use of the WLCG resources and middleware. In addition, it develops and utilizes its own VO-specific software. The following are some of the components used by CMS:

- **ProdAgent/WMAgent**: Workload systems for centrally-managed data processing and event simulation. *ProdAgent* used WLCG's WMS to submit jobs to sites (or local resources) [26]. The early work on this thesis interacted with ProdAgent, as discussed in Chapter 7. ProdAgent was replaced by *WMAgent* [27], which was also able to schedule jobs to sites directly but nowadays has been moved to a late-binding overlay model, using *glideinWMS*.
- **CRAB**: CRAB (*CMS Remote Analysis Builder*) is the tool used by physicists to execute their analysis programs on the grid [28]. CRAB interacts with the local user environment, the CMS data management services and the grid middleware. Its goal is to hide the complexities of the grid environment from the user in order to ease their work. It currently uses glideinWMS to schedule jobs at the sites.

- **glideinWMS**: A pilot-based late-binding overlay system [29]. We will discuss this in Chapter 3.
- **PhEDEx**: Data placement and transfer system [30]. PhEDEx manages the location of all CMS data files and their transfer between storage elements. In order to deal with the different storage technologies (including tape access), a series of PhEDEx agents run locally on each site and are configured by local administrators. These agents communicate with the central services, which coordinate the datasets ownership, subscription, transfer, etc.
- **DBS/DLS**: The DBS (*Dataset Bookkeeping Service*) and DLS (*Data Location Service*) are databases containing metadata and semantic information (physical meaning) of CMS data files, as well as providing a look-up service for workload management systems to locate data [31].

2.2. The Cloud

2.2.1. What is *the Cloud*

According to the National Institute of Standards and Technology [32], *cloud computing* is:

> *A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

Cloud providers offer remote access to large pools of (usually, virtualized) resources upon request (usually, for a fee). Even if variations of similar paradigms have existed since the time of the mainframes, the modern concept of cloud computing was popularized by the success of the *Amazon Web Services* offer, launched in 2006 [33]. As we already indicated, cloud computing can be considered to be a new materialization of the *utility computing* paradigm: instead of owning computing resources, users consume cloud services (ignoring where these resources are actually located). Cloud providers claim that economies of scale make their offers cost-effective. They rely on virtualization technologies to achieve the required flexibility to comply with consumer needs and the elasticity to adapt to the demand dynamically.

Several different cloud computing service models exist. The three most often cited are:

- **Infrastructure as a Service**: The consumer is provided with the capability to deploy fundamental computing resources (processing, stor-

age, networks) and run arbitrary software (including operating systems) on them. This is usually achieved by letting the user ship its own VM (*Virtual Machine*) to the underlying cloud infrastructure (which is out of the consumer's control).

- **Platform as a service**: The consumer is provided with a framework (including a number of libraries, tools and services) where she can deploy her application (with no control of the underlying environment or operating system).
- **Software as a service**: The customer makes use of remote applications through their provided interfaces (often through a web browser). The user does not have any control of the environment or the resources where these applications are run.

It is arguable whether cloud and grid computing are the same thing or one is a subset of the other or, even if they are completely different models. In our view, they are closely related paradigms, especially because both of them aim for the consumption of computing resources as utilities, accessing them at any moment and from any location, but there are also some differences between them, as they are generally understood. This is also the view of Foster *et al.*, who provide an in-depth comparison of both technologies [34].

Possibly, the main difference between the grid and the cloud is philosophical: while the grid vision was to federate (share) resources from different collaborators and to enable the access to any kind of resources through standard interfaces, the cloud, originated in the private sector, was presented as a centralized and closed offering of services and mostly in the form of proprietary solution. Technologically, the cloud, more recently born, has raised the scale of the resource centres and has embraced virtualization to ease the management of resources and to gain flexibility. In addition, cloud providers have been more successful than developers of grid tools in offering easy-to-use services and have attracted a much wider community of consumers. This can be easily observed in Figure 2.2, which shows the different volume of searches in the Google search engine for the terms *grid computing*, *cloud computing* and *big data* (which will be dealt with in Section 2.3). It can be seen that grid computing is almost forgotten by now while cloud computing reached higher levels of popularity, even if this are already decreasing.

But some of these differences are starting to blur because both models are evolving. Private cloud products are now maturing. These apply the dynamic management of resources of commercial offerings but are run by customers themselves (avoiding certain privacy concerns) and employ open interfaces. In parallel, virtualization is also being incorporated into the management of resources in grid centres. In fact, private and public clouds can also be seen as a new abstraction, part of a more complex grid of federated and paid resources. In the end, the important point is that users (in particular

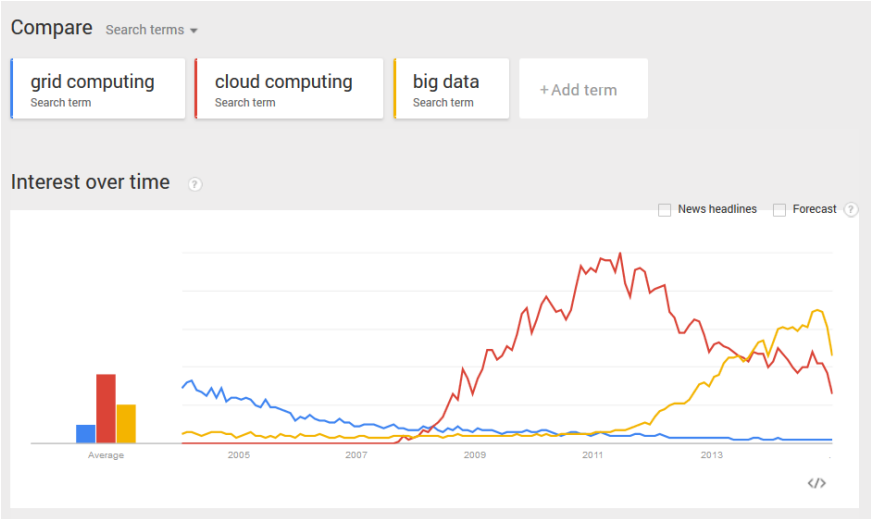


Fig. 2.2: Search volume for *grid computing*, *cloud computing* and *big data*. Source: *Google trends* (`http://www.google.com/trends`), accessed on January 4th, 2015.

researchers) have nowadays the technology—be it grid, cloud or a mix of them—to access a vast amount of resources, enabling them to execute a new range of applications.

2.2.2. Cloud Computing for Scientific Research

We have seen that, while grid technologies failed to gain wide acceptance outside of the academic circles, they are still consistently used in real scientific workflows today. The EGI and OSG infrastructures and, especially, the WLCG are the foremost examples of this. With the irruption of cloud computing, the scientific community has started to evaluate if they can also take advantage of the vast resources made available by cloud providers. In addition, clouds offer clear advantages over traditional (grid) data centres. In the infrastructure-as-a-service model, scientists can ship a customized VM with all the necessary software and configuration to run their experiment. This eliminates the burden of adapting the scientific applications to different environments and reduces the risk of misconfiguration errors. The second desirable characteristic of cloud resources is that they can grow or shrink on demand. This means that scientific groups can undertake costly computations as soon as the need arises, without having to wait for their associated data centres to provision new resources [35].

However, while cloud technologies are designed to be easily used by the average company or consumer, it is not clear that they can accommodate all kinds of scientific applications, which sometimes have demanding requirements. In particular, while clouds offer virtually unlimited processing power, their data management capabilities (providing efficient access to large storage systems and making use of data location information for scheduling) are still limited [34]. In addition, the virtualization of resources imposes certain performance penalties, especially in I/O (*Input/Output*).

Another important criteria to consider when evaluating cloud usage is, naturally, the monetary cost. The charges associated with moving data in and out of the cloud are currently prohibitive for data-intensive workflows [2]. Moreover, even for CPU-intensive workflows, it may be more convenient to acquire the necessary resources than to routinely run in the cloud. Early studies on the usability of cloud computing for scientific computations have yielded negative conclusions [35], also in the particular case of CMS [36]. However, even if they do not recommend using the cloud for the baseline operations, they concede that such on-demand resources may be very helpful to scale out during times when spikes in usage are required.

Although existing cloud offerings have not been deemed appropriate for all scientific applications and, in particular, an infrastructure like WLCG will not be abandoned anytime soon, the scientific community is very interested in cloud technology [1]. First of all, because they aim to be prepared to use the cloud resources, should the necessity arise, their cost be reduced or their capabilities improved (since they are still maturing) [36]. Moreover, the community is also trying to incorporate some of the advantages of the cloud into their own infrastructures. For this, they are building private clouds, where resources are virtualized and access for the scientists is simplified (e.g., CERN's agile infrastructure [37]). In summary, the experiment frameworks must be prepared to deal with grid and cloud and possibly other future resources and services. It is unlikely that a uniform set of standard interfaces will be available soon, so these frameworks must be able to deal with a variety of them.

As we will see, most of the work we present in this thesis, even if it was developed and tested in the grid environment, is equally applicable to the cloud paradigm.

2.3. Big Data

The general trend of massive data production and the technological capability to capture, store and process it have been identified as an opportunity for business, science and society. By applying advance statistics and analytics, data mining and machine learning techniques, powerful insights may be obtained. Business aim to understand customer patterns better, to

be able to measure the impact of their sales or marketing strategies, or to develop more accurate risk calculations. Scientific research, as we have seen, relies more and more in data coming from telescopes, medical imaging or environmental monitoring. Finally, customers enjoy new applications, from recommendation engines and price comparators to automatic language translators.

In this context, the term *big data* has emerged as a reference to this current abundance of data and the opportunities its analysis offers. Technically, the term is often used to denote any collection of data sets that are too large or complex to be processed using traditional applications. To be more precise, the community usually identifies three main characteristics for big data, these are the so-called *three Vs*: volume (large data sets), variety (different types of data, usually unstructured, semi-structured and structured) and velocity (high frequency data generation, e.g., via streaming or real time applications) [38, 2].

2.3.1. It is the Data

The big data phenomenon has drawn great attention in the research community, the industry and the media. This has been identified by Gartner in its latest *hype cycle*, from August 2014, shown in Figure 2.3 [39]. We can observe that big data is still in the upper part of the plot (*inflated expectations*) while cloud computing is in the lower part (*trough of disillusionment*) and grid computing has even disappeared from the plot at this time. As Gartner's classification implies and as it occurred with the cloud and, earlier, with the grid, big data is now subject of certain hype. It is difficult to say what the impact of this trend will be when we look at it a few years from now. It seems clear, however, that the superabundance of data is a firm trend and that, if anything, it will increase in the future.

As we have seen, the scientific domain is not an exception to this trend. Handling large amounts of data is, more and more, a must for scientific applications. Actually, we can probably say that science preceded business or general society in this tendency. What is interesting, though, is that this need for technologies that are able to deal with huge data volumes has extended to the industry. Even if there may be peculiarities in the requirements of the research community and in their computing workflows, mainstream technologies are now concerned with problems that look similar to theirs [3]. Scientists are looking into industry solutions to tackle their problems, mainly for the obvious advantage that such solutions do not require dedicated development and maintenance efforts from the community itself (often subject to uncertain funding) [18]. We already saw this with the exploration of cloud computing usage but it is also happening elsewhere [40].

Another important implication of this data profusion is that, in many cases, the computational workflows become data-centric: processing is no

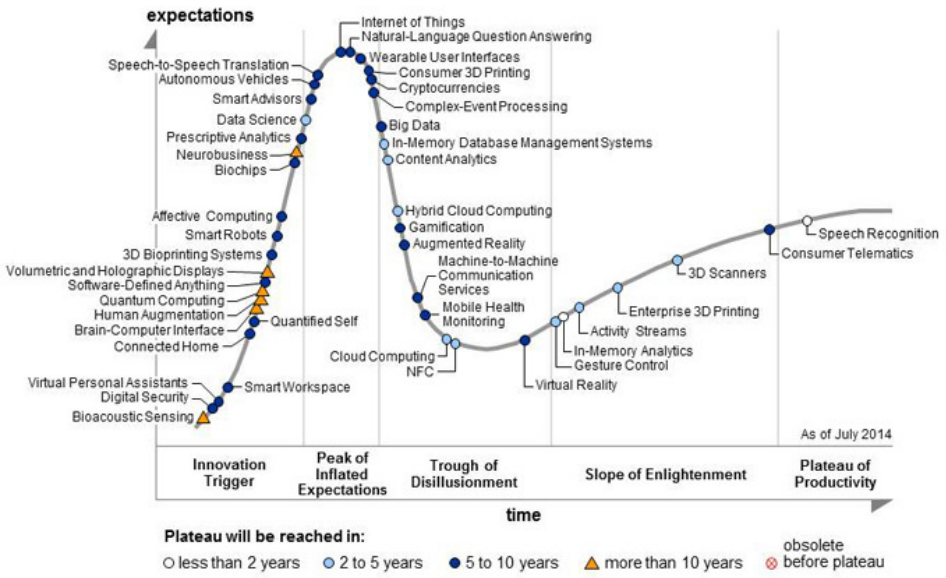


Fig. 2.3: Gartner's hype cycle for 2014. Source: `http://www.gartner.com/`.

longer the only issue, probably not even the main one; it is the data. Ensuring that the data can be properly stored and accessed, and optimizing scheduling to avoid excessive data replication or movement become fundamental duties for efficient workflow execution (time- and energy-wise). This was long ago realized by the grid community [41] and it is being acknowledged by the industry also, with data location-aware architectures, such as MapReduce [42] (and its later open source versions), and even proposals for new data-centric computing paradigms [43].

Wrap-up: *We live in the data age. Business, science and citizens are struggling to capture, transfer, store and process the torrents of data at our disposal. We have reviewed two of today's main large-scale computing environments, used precisely for those purposes. First, grid technologies: grown in academia and heavily used by substantial scientific communities. Second, cloud computing: general-purpose utility computation, impulsed by the private sector, but more and more used for the execution of scientific workflows. We have also provided a more detailed view of the currently most extensive grid infrastructure in the world, the WLCG, and of one of the four LHC experiments making use of it, CMS, whose needs and challenges have motivated our work. Finally, we have briefly discussed the present excitement around the big data concept and what*

this may bring, in terms of both opportunities and computing paradigms.

Efficient scheduling and execution of data-intensive workflows is the central problem addressed by this thesis. While our work originated in the specific context of the WLCG and targeting the optimization of the CMS experiment workflows, we argue that the issues we face and the solutions we find are of more general applicability. For one thing, scientific applications are incorporating mainstream technologies and industry solutions, such as the cloud and new big data tools. Also, industry and society are moving towards a world of data abundance. Appropriately handling this data and developing workflows that can consume it efficiently are becoming major requirements for modern computing deployments in many different domains.

Chapter 3

Workload and Data Management

The duty of deciding when and where computational tasks are run, and controlling their execution, is called *workload management*. This involves discovering which processing resources are available, selecting the best ones for a given task, dividing the task into a series of batch jobs, dispatching these jobs to the resources, watching the evolution of their status (making sure they are run successfully) and, finally, recovering the output produced by them.

While individual users can employ the grid protocols and tools to gather resource information and submit computing jobs to the appropriate destinations themselves, this task becomes complex and tedious when the number of tasks and resources increases. Most usually, the responsibility for resource selection and job submission is given to a *metascheduler*. The nominal metascheduler for EGI and WLCG was called WMS and followed the traditional grid brokering paradigm. This service was considered a constituent part of the infrastructure and could be used by all the grid VOs. However, the system has been gradually abandoned and replaced by VO-specific late-binding overlay systems. Both the traditional and the late-binding systems are discussed in Section 3.1.

Even if the initial grid proposals were perhaps more concerned with enabling the access to distributed processing capabilities, it was soon realized that the *data* was a very important resource on its own right. Moreover, given the increasing sizes of the data volumes handled by some applications (especially, in the scientific domain) and the heterogeneity of requirements, technologies and policies, it became evident that its management would be far from trivial. Early on, the concept of *data grid* was introduced as a general architecture that would enable different data-intensive applications to properly store, replicate and access data, manage metadata information, provide look-up services and co-schedule data transfers and computation [44]. The

WLCG architecture was designed to provide these but practice has shown that the task was really complex and challenging. Section 3.2 discusses the WLCG data management services and the problems it faces.

Finally, we will focus on the scheduling of applications consuming large amounts of data. It has been long accepted that, within a computer system, memory management and process scheduling should be considered in a coordinated fashion [45]. A similar reasoning may be applied to distributed scheduling of data-intensive computational tasks, at several levels (grid-wide and within resource centres). Section 3.3 analyzes this question and reviews traditional and current proposals while Section 3.4 discusses modern approaches to the problem, considering both mainstream scientific computing trends.

3.1. Workload Execution

Usually, the main objective of any scheduling system is to improve the efficiency of the workload execution. The most straightforward way to understand *efficiency* here is the aim of minimizing the time elapsed from job submission to retrieval of the output (*job turnaround time*). This can be extended, however, to the time required to complete a given workflow[1] (*workflow turnaround time* or *makespan*) or, perhaps, the average completion time for all the jobs of a user or a VO, or even all the jobs that are run on the grid infrastructure (*global efficiency*). As we will see, these targets are often compatible with each other, but, in occasions, they may be conflicting. This kind of difference is noted by Buyya *et al.*, who compare traditional cluster schedulers and grid brokers [6]. They state:

> *The schedulers in cluster systems focus on enhancing the overall system performance and utility as they are responsible for the whole system. On the other hand, the schedulers in Grid systems called resource brokers, focusing on enhancing the performance of a specific application in such a way that its end-user's quality of service requirements are met.*

In principle the aim of a distributed computing system should be to optimize both criteria, to the extent possible, or, at least, reach a certain compromise solution.

3.1.1. Traditional Grid Brokering

The traditional model for job scheduling in the grid is outlined by the diagram in Figure 3.1. Users send their computational task requests to one

[1] By workflow, we refer to a collection of computational jobs, which may be related to each other via dependencies (i.e., requiring that some jobs are executed before another one is started).

of the existing metaschedulers (WMS), which will in turn submit them to some of the computing elements in the infrastructure. The WMS must select a target CE for each job and it does so based on the information it receives from each possible destination and the preferences of the job submitter. The resource information includes policy and configuration (e.g., which VOs are authorized to run there, how many slots are allocated to each one, etc.), status (how many jobs are running or queued on the site already) and others (e.g., whether certain site holds the data required by the job at hand or if some software is installed). The CE passes the received job to the batch system, which enqueues them and, eventually, schedules them to run on some of the nodes of the computing farm.

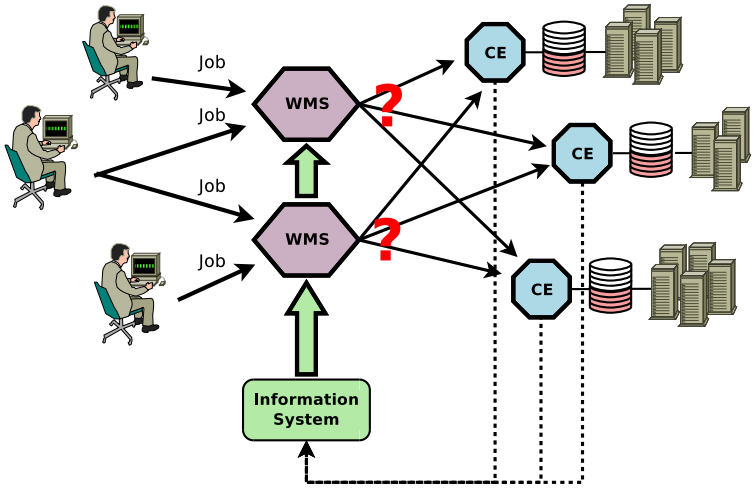


Fig. 3.1: Traditional scheduling of grid computing jobs.

The submitter preferences are indicated by the client via the *requirement* and *rank expressions*. The first one is a logical expression that indicates a series of constraints that the resources must satisfy in order to be eligible as destination (e.g., that the site holds some required data or that it is located within certain domain). The second one is an arithmetic expression used to order those matches according to their ranking value (e.g., prefer those sites with higher number of free slots).

The described model was dominant in WLCG for a long time since it was relatively simple to operate and satisfied the baseline requirements. The distributed nature of the WMS services (which operated independently) allowed the architecture to scale as desired by just deploying more instances of the brokers. However, extensive experience with the system, at large scales, revealed that it suffered from several problems. The two most important ones were its absolute reliance on the information system accuracy and the

existence, in practice, of two scheduling levels, one at the WMS and another one at the site's batch scheduler [18, 29].

Regarding the information system, there are several factors to consider. Firstly, in grids like WLCG, the amount of available resources is so large that it is not realistically possible to have an accurate view of the status of all services at all times. A minimum limit must be set for the information polling period. However, it is not unusual that the status of a site varies drastically in a short period of time. Since brokers work independently, it may happen that several of them detect that a site has free slots and decide to send a fairly high number of jobs to it. The CE of the site may then become overloaded by the combined amount of jobs from all the WMS's. Moreover, the grid information system often just fails to capture all the complexities of the site's batch system status and configuration. A site may configure different priorities and limits for different VOs or even types of users (*roles*) within each VO. In practice, it is not really feasible for the WMS to determine whether the job at hand will find a free slot at a given site or if it will have to wait on the CE's queue for long.

The second problem is the difficulty that VOs have to enforce their priority policies. When different jobs of a VO are queuing at a given site, it has no control of which ones run first. This is decided by the batch system's scheduler. WLCG VOs keep communication channels with sites and do provide instructions for configuration of the schedulers. However, since they have no direct access to the sites, any change requires manual intervention of the local administrators. With tens of sites supporting the WLCG VOs, this kind of actions may take weeks to be accomplished.

In addition to these deficiencies of the traditional scheduling mechanism, workload execution in grid environments suffers from other generic problems. Firstly, in order to profit from all existing resources, several submission interfaces must be used and grid jobs must be prepared to deal with heterogeneous environments, complicating the application development. In addition, given the size of the infrastructure, the chances that some WN causes job failures, because of a misconfiguration, a hardware breakdown or a disk becoming full, are quite high. Thus, the success rate for grid job execution is severely penalized. In extreme (though, not rare) cases, a single ill WN (but regarded as healthy by the batch system) causes the failure of hundreds of jobs because it constantly appears to be free and pulls new tasks, which die immediately upon execution start. This is called *black hole effect*.

Alternative job scheduling systems have been also proposed. For instance, auction-like systems where each resource bids for a given job request have been described. Calana is an example of this [46]. Its authors argue that this system eliminates the dependence on a global information system that unavoidably holds some old information (gathered at some time in the past).

However, auction-like systems have communication problems of their own, since job proposal have to be distributed to all resources and bids must be gathered and compared [47]. We will not discuss this kind of systems since they have never reached wide acceptance in large scientific grids, such as WLCG.

3.1.2. Late-binding Pilot Overlays

In order to alleviate the problems of traditional grid scheduling, some WLCG VOs developed and progressively adopted a model of *late-binding metascheduling* for their computing jobs. This model has proven very successful and has become the *de-facto* standard for the whole WLCG [18]. In late-binding scheduling, work is assigned to a node at the last possible moment before real execution. A VO agent is initially scheduled as a normal grid job (using traditional WMS/direct submission) with the main duty of *pulling* a real job from a VO task queue once landed on the execution resource. This VO agent is usually called *pilot job*. This model is illustrated by Figure 3.2. Even if designed for the grid, pilot job systems have been lately used to successfully integrate cloud nodes and even volunteer computers into the VO resource pools [36, 48].

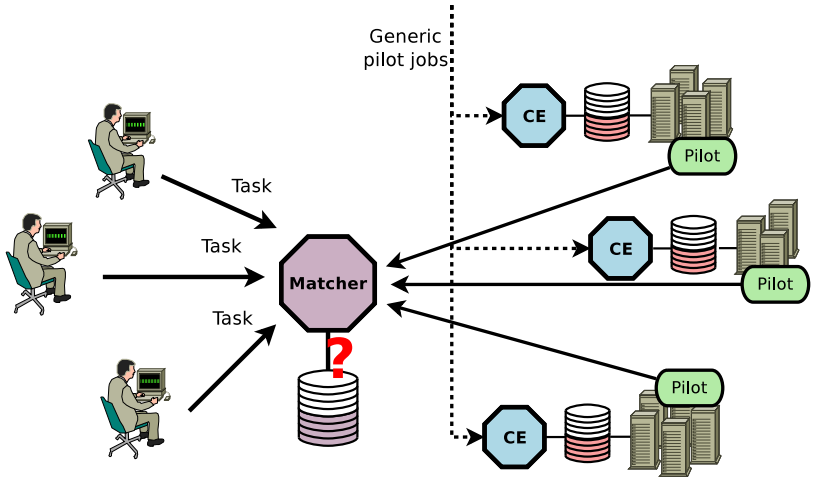


Fig. 3.2: Late-binding task scheduling using an overlay of pilot jobs.

There are several advantages in the use of pilot jobs. Firstly, pilots are sent to all available resources, using different interfaces, but present an homogeneous environment to the real applications (which can thus be made simpler). They also verify that computing resources are healthy (according to VO-specific criteria) before real tasks are run, thus significantly reducing failure rate (and avoiding black holes). In addition, late binding means that

pilots request tasks only when an execution slot is available. Therefore, uncertainties about available resources and waiting times greatly decrease. I.e., the architecture does not depend on obtaining totally accurate knowledge about status resources from the information system, as traditional scheduling did. Finally, since jobs are retrieved from a unique queue of tasks, which is directly managed by a VO, the global priorities can be adjusted almost immediately (with no manual intervention at the sites).

The described advantages justify the success of the pilot systems in WLCG. Nevertheless, their use brings an additional desirable effect: the reduction of the makespan of grid workflows. The late-binding approach offers better performance (reduced completion time) for the fulfillment of finite computational workloads. This effect is discussed in depth, with the help of an analytical model of grid workflow execution, in Chapter 5.

We are very interested in the late-binding overlay systems because we have relied on this model to build our own architecture for workload execution in distributed environments. We profit from the benefits that this paradigm provides and enhance it by adding new functionalities, namely *data caching* and *micro-scheduling*. This new architecture is introduced in Chapter 7

3.1.2.1. Existing Pilot Systems

Pull-based overlay architectures, similar to the WLCG pilot systems, were found earlier in the literature. An example of this is the *placeholder scheduling* [49]. Moreover, the pull-based paradigm can also be seen as a generalization of the well-known master-worker pattern. In particular, Berthold *et al.* propose a hierarchical version of this paradigm, which resembles the task delegation to the site's pilots occurring in our proposed architecture [50]. However, this is the only resemblance point since, within their subclusters, centralized task scheduling is used while we use a distributed matching procedure.

DIRAC (*Distributed Infrastructure with Remote Agent Control*) was the first pilot-based system used by a large VO (LHCb) in WLCG [51]. Since then, several other VOs have adopted the same model. Examples of large-scale systems in use today are AliEn (*ALICE Environment*), used by the ALICE VO [52], PanDA (*Production and Distributed Analysis*), by ATLAS [53], and glideinWMS, used by CMS [29]. All of these systems share the main architecture model: pilot jobs are sent as placeholders and they retrieve real tasks at runtime. The first three systems use a similar central queue, accessed using standard protocols, such as HTTPS or secure XML-RPC, from where tasks are pulled. As for glideinWMS, it is an extension of an HTCondor system, in which there are two *Collector* daemons (responsible for matching jobs an resources), one at the *factory*, which deals with pilot jobs and one at the VO front-end, which matches the real VO tasks with the

resource descriptions sent by the pilots [19].

The functionalities offered by these systems and the lessons learnt by their VOs serve as a major inspiration for our work. Some interesting characteristics that we have taken from them (in general, introduced by DIRAC, then adopted by others) are the following:

- Pilots send periodic heartbeat messages so that the central servers can notice if the pilot dies for some reason and if the task it was running must be rescheduled.
- Each pilot runs several tasks in a row, as long as it still has enough time before the limits imposed by the batch system are reached. In this way, the overhead incurred by going through the grid submission process and the batch system job start is divided among the number of tasks run on the pilot.
- VOs keep a certain excess of pilot jobs (with respect to enqueued tasks) to cover inefficiencies of real computing resources or middleware.
- Submitted pilot jobs are as lightweight as possible. Any additional required libraries are downloaded from a web server in order to avoid congestion of grid submission systems and to allow the use of standard caching procedures for the libraries.

3.1.2.2. Issues and Challenges

Even if pilot systems have proved very successful, they also present a few issues and face some challenges. The first one is that—born within the VO frameworks—they were not designed as general-purpose systems, but optimized for the VO software and needs. In the last years, however, there have been strong efforts to revert this situation and make the systems usable by other VOs. In particular, we know that DIRAC is being used by several smaller VOs (which lack the funds to develop another pilot system of their own) but this is not the only move in that direction.

Another problem is that of traceability. Traditional grid jobs must authenticate with the certificate of their submitter. Thus, it is always possible to map a job with the individual who sent it (which is a security requirement by WLCG). Pilot jobs, on the contrary, are submitted by central services and can run jobs of any user, so, in principle, it is not easy to trace from a problematic task to its submitter. This has been partially solved by the use of *gLExec*, used to perform an identity change from the pilot to the user associated to the real task credentials [54]. However, this technology is complex and problems have been found when trying to enforce it in all WLCG sites [18].

Finally, all of the indicated pilot frameworks rely on a centralized task matching and assignment procedure. Apart from the fact that a central queue of tasks may become a SPOF (*Single Point of Failure*), as the number of pilots and tasks increases, the process to match real jobs to pilots gets harder, leading to potential bottlenecks in the queues. While the pilot systems of the WLCG VOs seem to satisfy their needs, this might not be the case if finer matching requirements were imposed. This is discussed in more detail in Chapter 8.

3.2. Data Management

The European research projects preceding and up to EGI aimed to raise an international data grid infrastructure for e-science. They built on the concepts proposed by Chervenak, *et al.* [44] to define the main data management services supported by the infrastructure—storage elements, file transfer services and file catalogs—as indicated in Section 2.1.3.1. However, the materialization of this architecture into real computational services, usable by the scientific VOs, proved to be even more challenging that foreseen. It took several years before the middleware provided by projects like WLCG was capable of delivering the expected functionality at the required scales, with affordable operational costs [55]. In fact, the grid data management technologies are still—and probably will always be—evolving.

Let us now briefly review some of the main data-related issues confronted by WLCG VOs and how they are tackled.

3.2.1. Storage Elements

Storage elements are the basic building block of the WLCG data management system. They hold all the data produced, in one way or another, by the experiments. Data must be kept (sometimes for a relatively short lifetime, in other cases, for decades) but it also must be accessible for a high number of users and computational tasks (with varied access patterns and using different protocols), and it must be easily replicable to other SEs, through the WAN (*Wide Area Network*). Essential functionalities that a SE must support include transparent capacity growth or shrinkage, namespace management, enforcement of complex authentication and authorization policies, space reservation and automatic file removal, if necessary. But the list of requirements does not stop there.

Since data must be efficiently accessible by batch jobs and the number of concurrent clients may be high (up to thousands on a single grid site), large disk pools are customarily used as front-end technology and they are carefully networked to the WNs on a site. Still, the wide range of different activities performed by WLCG VOs, the complex access patterns that are

often employed by the experiments and the need to also serve data replications to remote clients (high-latency, sequential, intensive copies) have sometimes led to congestion problems when accessing SE's data [56, 57]. These problems have of course increased when the number of I/O intensive jobs in a site is high and, even more so, if an elevated number of clients try to access the same file (or small set of files), causing what is called a *hot-spot*. In order to fight this problem, VOs routinely set limits to the number of I/O bound jobs that can simultaneously run against a single SE and the storage systems apply techniques such as automatic replication of highly accessed files.

But, in addition to disk pools, part of the WLCG data is archived in tape, for cost efficiency and data preservation reasons. Tape facilities are integrated into tertiary storage systems, which automatically archive data after some expiration time and stage it back to disk when accessed again. Because of this, SEs must also provide pinning capabilities, so that VOs can mark data that should not be archived because it will be used soon. Moreover, in order to avoid excessive data movement between disk and tape and to ensure that no data is lost, WLCG defined a set of *storage classes*, which indicate what files should be kept in disk, in tape or both [23]. This depends on whether the site is the final responsible for the data (or some other site is), and whether the files are expected to be accessed often.

At CERN, WLCG's Tier-0 and largest site, the number of performed activities and access patterns, as well as the number of data consumers (users and jobs), are multiplied, and the management of a single SE becomes more and more complicated. That is the reason why EOS, a disk-only modern SE with in-memory namespace [58], was introduced to complement Castor, the original tertiary storage system at CERN [59]. While Castor was designed to meet the requirements of central data recording (fewer concurrent users, organized data access, mostly write-once data, transparent tape migration), EOS is optimized for read-write disordered analysis, with low-latency requirements, and interactive access of hundreds of users.

But even if the demands set by WLCG VOs to the storage services are challenging, the grid SEs have gone a long way to reach the required functionality, robustness and performance. Thanks to continued technological advances, today's SEs are more capable to fulfill their duties. In addition, site's internal and external networks have been improved, storage systems have been rationalized (like in the described Tier-0 example) and important optimizations have been applied to the processing applications, meaning that more efficient data access can be now performed [56].

3.2.2. Data Distribution

We have described the systems that store and serve data. However, storage space is not infinite and data transferring is very resource consuming (in

terms of time, network and load). A thoughtful model for the distribution of data in sites and an efficient service for organized data replication are essential for a grid infrastructure like WLCG.

In regard to data distribution, experiments must decide how many times each dataset should be replicated, where these replicas should be stored, if they should go to disk or tape and so on. This is considerably complicated by the fact that WLCG VOs have traditionally coupled computing jobs to data locality (more on this, in Section 3.3), which means that highly accessed datasets should be replicated to more sites (and also be available on disk) and that sometimes data transfer requests cannot be predicted in advance (since they depend on the activities of analysis groups and individual physicists). Moreover, if significant portions of archived data must be reprocessed, centrally coordinated pre-staging of the data must be carried out beforehand.

As for data movement, a dedicated DPS (*Data Placement System*) is used to ensure that datasets get copied to the intended destinations. If we take the example of the service used by CMS—PhEDEx—sites just get subscribed to datasets and PhEDEx takes responsibility for finding the sources for the data, instructing the appropriate endpoints to initiate the file transfers, watching their progress and performing the necessary retries on failures (thus greatly reducing the necessary human intervention). For those datasets that may accept newly produced files, PhEDEx also takes charge for copying the new data when it is available. Moreover, since DPS's have a global view of data distribution and movement, these components can optimize network usage by appropriately grouping transfers and can protect the SEs by limiting the number of concurrent transfers that are scheduled on them. While individual members of the VOs are given the ability to manually perform file replication, massive transfers must be sanctioned by authorized operators and, once approved, they are managed by DPS's.

Other issues found when dealing with massive and complex sets of data were the needs to perform frequent consistency checks to ensure that the file catalogs are aligned with the real contents of SEs and the obligation to undertake organized pre-staging of tape files into disk pools before scheduling reprocessing tasks. At any rate, all the indicated tasks—data distribution, supervision of transfers, ensuring consistency and coordinating the various centralized activities–still require considerable amounts of operational effort. For this reason, the VOs are now in the process of automating some of these tasks, as we will discuss in Section 3.4.2.

3.3. Data-intensive Scheduling

A distinctive characteristic of data-intensive workflows is, by definition, their need to efficiently access large volumes of data. If this is not a concern,

then they should not belong to this category. When scheduling jobs in a grid infrastructure, the primary worry is often to let jobs run in resource centres where required data is locally available. However, even when scheduling tasks within a single cluster, taking into account the location of the data in the individual nodes may be crucial to achieve satisfactory access throughputs. The following sections discuss both problems.

3.3.1. Grid Scheduling

It is generally admitted that file locality is an essential parameter for the scheduling of data-intensive grid applications. If jobs are submitted into any available computing resource, regardless of their data needs, costly transfers will need to be performed in order to move the data to where the jobs have been sent. Extensive evidence of this is found in the literature. For example, Cameron *et al.* show that optimum scheduling decisions can only be achieved when both the processing/queuing time for tasks and the penalty for data transferring are balanced in the selection process [60]. To compute these values, file catalogues must be queried and information on the relative network speed of the links between sites must be available. Efforts to incorporate these considerations into a scheduling system were made, e.g., by the DIANA (*Data Intensive And Network Aware*) metascheduler [61] and the Gridbus broker [62]. They select job destinations based on the estimation of computational power of resources and data transfer costs, profiting from a distributed network monitoring service. For each job to be scheduled, a handful of computing nodes are selected according to their power, queueing jobs and data transfer associated costs. In the case of DIANA, in addition, once the job's destination has been chosen, the best replicas of the required input files are selected.

However, such complex brokers have never been widely used in WLCG. The reality is that performing accurate estimations of the foreseen duration of task execution and input data transfer is usually not feasible in practice. Processing speeds are heterogeneous within resource centres and queuing times are not always predictable. Data transfer speeds depend on too many factors and no network weather service has been available in WLCG, at least until very recently. In fact, the scheduling policy supported by EGI's WMS and traditionally used by WLCG VOs has been a simple submission of jobs to the computing sites holding the required input data. We find a more sophisticated behaviour in the workload management system of the ATLAS VO, which, in order to assign tasks to a particular set of sites, may balance the disk space available at the SE, the locality of input data and the available CPU resources, and may request that the data management system transfers input files to the selected sites. However, this is only done for non data-intensive simulation jobs while physics analysis jobs are always scheduled to sites where the required files are already available [63].

Moreover, even if it was possible to reliably estimate the duration of job computation and data staging, there are other factors to take into account. For example, the available space at each of the involved SEs may prevent some transfers to take place or the required data may have been archived to tape. Another point to be considered is the overhead associated with the large number of individual data transfers that an integrated job scheduler may cause. It is more efficient to let specialized DPS's group and tune data transfers asynchronously [64].

A different approach on the whole problem was taken by Ranganathan *et al.*, who propose to decouple the scheduling of jobs and the replication of input data [41]. They suggest that an asynchronous replication of popular (most-accessed) files should be performed by an external agent and that the scheduler should just submit jobs to sites holding required data at the moment that the matching was made. They show that, under certain assumptions, this approach achieves better results than on-demand replication. They admit that this result may be different if technology offered greater WAN bandwidths. In any case, the great advantage of this model is that it simplifies the design of both subsystems and that it allows for independent optimizations of their functionalities. In fact, as we will see in Section 3.4, WLCG VOs have recently started to follow a very similar approach to the problem.

We also performed some early work on this topic, specifically enhancing an existing metascheduler to incorporate data location in its scheduling decisions. We will review this work in Chapter 6.

3.3.2. Cluster Scheduling

The collocation of code and data is a more general research topic, not at all limited to the problem of submitting jobs to the appropriate grid sites. This factor has been taken into account in supercomputing and local batch scheduling. For example, Korupolu *et al.* aim to optimize the placement of data and jobs in a cluster system by applying a new algorithm based on the similarities of the coupled problem to the *Knapsack* and the *Stable Marriage* problems [65]. They notice that this allocation problem is NP-complete[2] and that the use of a LP (*Linear Programming*) solver to find the optimal solution would be prohibitively slow. Their proposed solution yields satisfactory results in a shorter time (but still not too quick, in our view, since they report 333 seconds for the matching of 2,500 applications on 1,000 nodes).

Moreover, data locality is also an important factor being considered by modern commercial frameworks for parallel computation, where local file

[2]NP (*Nondeterministic Polynomial time*): no polynomial time solution is known for the problem.

access is the key to increase data access throughput and, thus, system's performance. Notably, the successful *Hadoop* framework heavily relies on this idea [66]. The Hadoop project was inspired by Google's *MapReduce* programming model [42] and the *Google File System* [67]. Like MapReduce, Hadoop provides a framework to easily write fault-tolerant, distributed programs (using the *map and reduce* paradigm) and, like the Google File System, the underlying storage, HDFS (*Hadoop File System*), is a performant, scalable and reliable distributed file system, built on inexpensive commodity hardware [68]. In HDFS, files are split in fixed-size *chunks* and distributed over the disk servers. Each chunk is replicated several times with the aim of increasing data reliability, availability and throughput. Hadoop takes advantage of HDFS by scheduling its *map tasks* on the nodes containing the data they need, thus allowing them to perform local reads and greatly reducing the network usage.

The need to use the MapReduce model to run on Hadoop has prevented some existing applications from using it: firstly, they need to fit the paradigm, secondly they would have to be rewritten to use it. This is the case for most WLCG applications, whose gigantic code cannot be easily modified. However, HDFS has been made to work as a site SE, accessible by grid jobs [40]. Interestingly, the authors moan about the inability of traditional batch systems to make use of HDFS locality information. They indicate that only 2 % of the jobs run on the nodes holding their required input data.

In fact, data locality has not generally been an issue for job scheduling within grid resource centres, since computation and storage nodes have been traditionally kept separated (WNs and SEs) and high throughput has relied on capable networks and the deployment of parallel storage servers. Research on grid scheduling has mostly focused on site selection and left intra-site job allocation to traditional (and usually *data-location unaware*) batch schedulers. We can find only a few works that study the job and data placement problem within a grid resource centre. Two of them modify HTCondor to include data locality and transfer needs into their scheduling process [69, 70]. The main target of the first of these works is to minimize data movements while the second one seeks to improve the overall planning of workflows.

These grid proposals show that data-aware intra-cluster scheduling may achieve workflow turnaround improvements by increasing tasks and data collocation and reducing data movements, especially given the current trend of cluster size growth. However, at large scales, they suffer from scalability problems since they are centralized solutions, where a planner ambitiously tries to workout the optimal data and job distribution. In addition, these algorithms rely on a specific site scheduler deployment and are not general enough to include other parameters beyond task lengths and data location

in the optimization process.

Our late-binding architecture also incorporates data locality in the scheduling of tasks within a given cluster in order to reduce network communication and protect site SEs by using a file cache. Our system is more general since it is integrated in the grid-wide pilot framework (superimposed on any batch system technology), can take into account any parameters beyond data location and is performed in a distributed fashion, thus avoiding scalability problems. This is discussed at length in Chapters 5, 7, 8 and 10.

3.4. Trends

3.4.1. Mainstream Products

Despite the great success achieved by Hadoop and the MapReduce framework, they are not the solution for all problems. In fact, several shortcomings are often imputed to Hadoop: setup complexity (especially for simple applications), slowness for iterative analysis, restricted application model (MapReduce), deficient support for multi-tenancy (in particular due to scalability problems by the job tracker when large shared clusters must be managed) [71]. Certainly, it is not possible to review the whole range of new products and technologies that aim to replace, enhance or complement Hadoop, or simply become alternative solutions for applications requiring to run on large distributed systems. For illustrative purposes, we will review a few popular and representative examples.

With the goal of solving the main issues of Hadoop, YARN (*Yet Another Resource Negotiator*), the new generation Hadoop's compute platform, introduces the resource manager and the AM (*Application Master*) [71]. The first one is a system-wide service that assigns a subset of the available resources to a given application. The AM is run by the application itself and is in charge of managing the different composing tasks. Each AM may implement any computing model; MapReduce is one example, but not the only possible one. In addition, YARN is more scalable, since instead of a single job tracker, there is an AM per application (similar to the hierarchical master-worker paradigm, already mentioned [50]). Even with YARN, Hadoop still relies on HDFS and the data location information it provides. The AM must query the resource manager about file locations before initiating the application. This procedure is much less dynamic that the one provided by our DHT-based distributed data cache.

Spark is a project that also addresses some of the fundamental limitations of Hadoop, namely the overheads for short tasks and the complexity required to perform simple analyses [72]. Spark is considered apter for iterative and interactive applications. In order to achieve these objectives, Spark loads most data in memory (though it still uses HDFS or another distributed

file system as back-end) and integrates its programming environment into a general-purpose language, such as *Scalla* or *Python*. Regarding scheduling, Spark uses a *cluster manager* to allocate *executors*, which are agents in charge of running the user tasks[3]. For the scheduling of application tasks, the Spark framework uses data locality (in memory or, failing this, on disk) and applies the idea of *delay scheduling*.

The *delay scheduling* was introduced by Zaharia *et al.* in order to overcome the conflict between fairness and data-locality matching in multi-tenant Hadoop clusters [73]. They started by replacing the default Hadoop's FIFO (*First in, First out*) scheduler by a fair-share queue. However, they noticed that, often, the next task to be scheduled (according to fairness reasons) could not be run on any node holding its required data. The simple delay scheduling algorithm tackles this problem by just delaying tasks for a limited amount of time so that they can run on a node with the data. They show that, given the size of their cluster, the multiple replicas of each data block and the short lengths of Hadoop tasks, a small amount of waiting time is enough to bring locality close to 100 %. Interestingly, we independently developed a similar strategy, though not identical since the contexts are different, for our late-binding architecture (see Section 7.3).

The Omega scheduler was developed with the aim of replacing Google's main monolithic scheduler [74]. The monolithic scheduler has complete control over the whole cluster and deals with all enqueued jobs. Even if highly customized and optimized, this scheduler faces several problems. Firstly, it is doubtful that it will be able to scale to even larger clusters (which are planned). Secondly, submitted tasks are heterogeneous and, while long-lived ones may tolerate relatively long scheduling delays (due to the complex requirements they impose), short-lived ones sometimes cannot accept such long waiting periods (before being matched). Finally, the monolithic scheduler is difficult to maintain and enhance without disturbing operations.

In order to deal with the described problems, Omega authors compare two-level schedulers (like YARN) and their proposal of shared-state, no-lock, competing schedulers. In the first case, a first-level scheduler distributes available resources among frameworks/applications, which, in turn, schedule their tasks within the assigned subset. This is regarded as rigid and pessimistic by Omega authors, who, instead, propose an architecture with different independent schedulers (per framework or type of job), with a global view of the system, and submitting jobs to all available slots. Only when a clash occurs, a scheduler reconsiders its job allocation. They describe this as an *optimistic approach*. It is remarkable how this approach resembles the traditional grid scheduling model of EGI and WLCG. There are however a couple of important differences. The first one is that Omega will operate in a highly controlled environment, where each scheduler can be limited or

[3] In that sense, the executors resemble the pilot jobs of late-binding overlays.

forced to satisfy certain policies. The second is that, even if Omega handles a large pool of resources, these will probably be contained within a single LAN (*Local Area Network*) and administrative domain, so their view of the cluster status will be far more reliable than that provided by the grid information system. Lastly, Omega schedulers are able to detect clashes with other peers and change their job allocation accordingly.

From the previous discussion, we would like to remark a few conclusions:

- Overcoming scalability problems and reducing scheduling delays is a major goal of most modern cluster schedulers.
- Coupling job allocation to the location of required data is still considered necessary to achieve high throughput.
- The discussed systems run mostly on single clusters only while applications on the grid access clusters on different sites (or clouds, etc.).
- The competition for resources among different tenants is a major headache for large setups and the problem is tackled in different ways.

The first two items are discussed at length in this thesis and they are addressed by our distributed matching architecture; please refer to Chapters 8 and 10. As for the third one, we must note that our late-binding architecture must deal with both the inter- and intra-site scheduling of jobs. Finally, regarding resource contention, we can find it in an environment like WLCG at two different levels. The first one, the competition among VOs, is not dealt with by our architecture (which is operated by a single VO). It is the duty of the different sites (and their batch schedulers) to ensure that each VO consumes its share of available resources. The second level refers to the contention among different applications or users within a single VO. In our system, other than the FIFO queue, this is handled by a simple priority mechanism. It is not clear whether WLCG VOs require more than this, but, admittedly, more complex patterns (e.g., fair-share enforcement or express queues) could be added to the architecture.

3.4.2. Scientific Computing

While the traditional scheduling policy of WLCG VOs has always been (and, to a great extent, it still is) to submit grid jobs consuming data only to the sites holding the required input datasets, this has been recently complemented by two additional mechanisms. One of them is the use of an automatic data placement system and the other one is the deployment of a global infrastructure for remote data access. This are discussed as follows.

3.4.2.1. Automatic Data Placement

We have already indicated that the operational cost of manually distributing datasets among different sites is very high. That is the reason why WLCG VOs have developed systems to automatically handle the replication and deletion of datasets, based on their statistical usage. This is well in line to the proposals made by Ranganathan *et al.* [41].

As an example of such a system we find ATLAS popularity and dynamic data placement services [75]. The first one provides summaries of data access per site, user, dataset or other attributes, and it is used to delete underutilized dataset replicas. The second one measures the instantaneous popularity of data based on the recently submitted user jobs and, if the number of replicas in the system is too low, it requests the creation of extra replicas. Notice that this implies that some of the queued jobs might be able to be submitted to the sites holding the new replicas when their time to run arrives. This also means that the workload and data management systems are not completely independent.

The CMS experiment has a similar popularity data service, whose statistics are used for automatic deletion and replication of datasets, based on age, size, total and recent usage. In addition, policies such as guaranteeing a minimum number of replicas for each dataset are enforced.

3.4.2.2. Remote Data Access

Several years of production activity have taught WLCG VOs that a collaboration of thousands of users cannot be easily contained within an excessively rigid model of operation. Taking the case of CMS as an example, the strict hierarchy of tiers, according to which, Tier-2 sites depending on a given Tier-1 should not transfer data from Tier-2s assigned to a different Tier-1, had to be broken in favour of a more flexible model where data can essentially flow between any two peers (though certain paths are preferred). Likewise, the paradigm of users accessing only local data prevents them from performing quick interactive analysis on data not held on their home institute and causes the failure of grid jobs if a single file is unavailable on the site where they are running[4].

To prevent these issues and, in general, to come closer to the grid metaphor of users being able to access data irrespective of its location, a data federation was developed in CMS [76]. This system defines a common namespace for all federated storage resources. The technology used to access this storage is *xrootd*, which defines the protocol to query for file existence and to access file contents remotely [20]. All sites must deploy xrootd servers and

[4]This effect is not uncommon. A SE holds a complete dataset and attracts jobs desiring to process it, but a few files may be corrupted or the disk holding them may be unavailable for some reason.

register them with global xrootd *redirectors*, which are able to point clients to the appropriate server holding their required data.

The xrootd data federation was initially used only for occasional access to individual files (mostly for interactive usage) and as a fall-back for grid jobs encountering data access problems at a given site. However, due to the success of the system, CMS is starting to apply it to other situations, such as diverting jobs requiring to process a dataset held only on an overloaded site or allowing remote processing on Tier-3 sites (with no storage infrastructure). Such practices are only possible due to the already mentioned work on CMS I/O optimization and to the improvements in WAN robustness and bandwidth [56]. Still, they are only suited for certain types of applications and imply some loss of efficiency. Moreover, care must be taken not to overload SEs and networks with too intense remote I/O. This leads to delicate policy problems: should all users be able to perform analysis on remote data or only central groups? Perhaps only for certain activities? This kind of questions are still being clarified by WLCG VOs.

WRAP-UP: *Workload and data management in large grid infrastructures, like WLCG, are complex and demanding tasks. Moreover, they are interrelated activities and any inefficiencies in one of them may affect the other. We have seen that this is an important factor not only for scientific grids and applications but for mainstream and commercial solutions. Technology and practices evolve to satisfy the changing needs of users and virtual organizations.*

We have reviewed the traditional and the more recent late-binding overlay architectures for grid scheduling. The first one suffers from excessive dependency on precise and up-to-date resource information and an inability to enforce VO priority policies. The second one solves these issues but still faces some challenges, of which, scalability will be of importance for our work. In regard to data, efficient access and adequate placement have been a concern since the inception of WLCG. Although great progress has been achieved, attempts to reduce the human effort required for the operation of services are still being undertaken. We have also surveyed the academic and commercial approaches to the execution of data-intensive workflows, especially regarding the coordination of data placement and job scheduling at a grid-wide level and within clusters.

The central work of this thesis is the proposal of a new scalable late-binding architecture, which builds on the existing pilot systems to provide new scheduling capabilities and improve data access by protecting massive storage systems and considering data-location at all levels. This will be discussed in detail in the following chapters.

Chapter 4

Distributed Hash Tables

A DHT (*Distributed Hash Table*) is a decentralized distributed system that provides routing and look-up services. A node belonging to a DHT is able to reach any other node given only its ID (*Identifier*). Moreover, the system can store data in a distributed way and any node can retrieve this data with the look-up operation. The absence of central coordination confers robustness (absence of single point of failure) to the DHTs and makes them a kind of structured P2P (*Peer-to-Peer*) network [77].

Another key feature of these systems is their scalability. As we will discuss later in the chapter, the properties of the DHTs make them able to reach millions of members without causing problems in the nodes or in the network. Both the robustness and the scalability make DHTs very attractive solutions for large distributed systems.

One of the most prominent examples of DHT application is that of the P2P file sharing systems. Networks such as *Gnutella* use a DHT to save and look up file storage locations [78]. Since, as we will discuss in Chapter 10, we will be using a similar system to share data among pilots in the grid, we need to understand the basic properties of DHTs and in particular of *Kademlia*, our chosen P2P system [79]. A discussion on these is presented in Sections 4.1 and 4.2.

Moreover, Chapter 10 also introduces a new distributed task matching algorithm based on the underlying DHT. This algorithm requires the utilization of the broadcast operation, which is not usually present in most DHT systems. As a result of this need, we have studied the problem of broadcasting in DHT systems and in Kademlia in particular. Sections 4.3 through 4.5 review related work on this topic. A more detailed discussion on the specific application of the broadcast operation to Kademlia, including our contributions on the subject, will be presented in Chapter 9.

4.1. DHTs

4.1.1. General Description

In a DHT, both the participating nodes and the stored data are uniquely identified by and ID (in the case of the data, the IDs are also called *keys*). Data IDs are normally produced by using a known hash function so that they can be easily computed from the data itself. Due to the properties of the hash functions, all IDs belong to a finite space of possible values. Each node in the DHT is made responsible for a subset of the ID space using what is known as *consistent hashing*, so that, when nodes are added or removed, only a small fraction of the keys must be reassigned [80]. How the assignment of the keys is made exactly is one of the defining properties of the different DHTs. When a new data element needs to be stored in the DHT, its key must be computed and the peer that should be responsible for that key is looked up. At this point, the node is contacted and the data stored there. When a client wants to retrieve the data, it only needs to perform the same look-up process again to find out which node holds the desired value and ask for it.

Nodes in the DHT maintain a table with the address and ID of a subset of the nodes in the system. An iterative (or recursive) algorithm must exist that allows any node to find the peer responsible of an ID with the only prior knowledge of its contact table. The general approach to tackle this problem is that a node must have good knowledge of the nodes whose IDs are *close* to his own one and keep a short list of contacts at further distances, in order to forward look-up queries to them, as necessary. Thus, the algorithm to search for a node with certain ID basically consists on iteratively contacting nodes that are closer and closer to the target. The definition of ID *closeness* (i.e., how distance between IDs is measured) is a key DHT characteristic that differs from one system to another.

It is important to note that, as we will see in the examples below, nodes in DHT systems are able to reach any other member in a limited number of iterations and with help of a very short list of contacts, even for networks with large number of nodes. As a matter of fact, the number of steps required to find a node grows normally with the logarithm of the size of the network [81]. This ability makes DHTs scalable.

Normally, when a node joins the system, it needs to contact an existing peer and then starts a procedure to discover other nodes. It is also common that other routines exist to keep the contacts table up to date and to perform any necessary rearrangement when a node leaves the network. The details of these procedures are also important properties of the distinct DHTs.

4.1.2. Applications

As we already indicated, DHTs were firstly applied to file sharing systems. Since then, however, they have been used as the basis for multiple other applications. Their limitless scalability and the absence of single point of failures make DHT systems the perfect candidates to serve as the look-up or overlay routing facility for any distributed application that involves large number of nodes.

DHTs have been proposed for many diverse uses, such as multicast and anycast delivery, distributed data storage, DNS (*Domain Name System*), search engine, CDN (*Content Delivery Network*), voice over IP communication, distributed database (especially noSQL) and more [82].

Focusing on proposals that are similar to the one presented in this work, we are interested in the use of DHT-based techniques for file location tracking. In this case, apart from the storage systems (e.g., PAST, based on Pastry [83]) and the file sharing services (e.g., Freenet [84]), we can cite *memcached* as the paradigmatic example of data cache using consistent hashing [85]. In the grid world, we find a P2P grid replica location catalog [86]. We do not add any contribution to these works. We just use the DHT for file tracking and sharing in a way that is similar to the previous proposals.

Our use of the DHT for distributed task matching is a different case. There have also been prior efforts to utilize distributed techniques for scheduling. SwinDeW-G is a workflow management system where grid nodes interconnected using P2P techniques coordinate to distribute computational tasks [87]. Likewise, Cao *et al.* suggest the deployment of local schedulers on resource centres and their cooperation using a P2P network [88] and Rahman *et al.* propose to create a DHT-based logical coordination space where grid resource and workflow agents can cooperatively schedule their workflow tasks [89]. However, in this case, our work does offer some contributions. The granularity of all the indicated proposals is at the site level, thus no micro-scheduling is performed, and their P2P networks are grid-wide while our DHTs are confined to a site's LAN. Furthermore, their models assume early-binding matching (even if they expect their systems to have more reliable information than traditional grid brokering systems). Lastly, none of these works show tests of the scale that we do.

4.1.3. Examples

There is a large number of DHT proposals and implementations [77, 81]. It is not possible for us to review even a significant subset of them. We will however discuss the main characteristics of Chord, since this was one of the first proposed DHTs and it is often used as a reference DHT example in the literature. We will also talk about Pastry because of its prefix-based routing

algorithm and its resemblance with Kademlia.

4.1.3.1. Chord

Chord uses a circular ID space of size N, where both nodes and keys are mapped [90]. Starting from 0, the IDs increase if we follow the circle clockwise. The node responsible for a given key is its *successor*, i.e., the node whose ID is found first when turning clockwise (it may happen that we reach the beginning and continue from 0).

A node with ID k has a pointer (*finger*) to its successor and the node preceding it, as in a double linked list. In addition, the nodes keep $M = log_2 N$ fingers to the successors of the IDs with values $k + 2^{i-1}, i = 1..M$. With this scheme, every node has fingers pointing to nodes that are distant from it by approximately half the ID space, a fourth of the space, etc. This is illustrated by Figure 4.1, which shows a Chord system with a small ID space of $N = 128$ and a few nodes (12, 29, 45, etc.). It also displays the finger table for node 45. In this case, its contacts are nodes 61, 87 and 114.

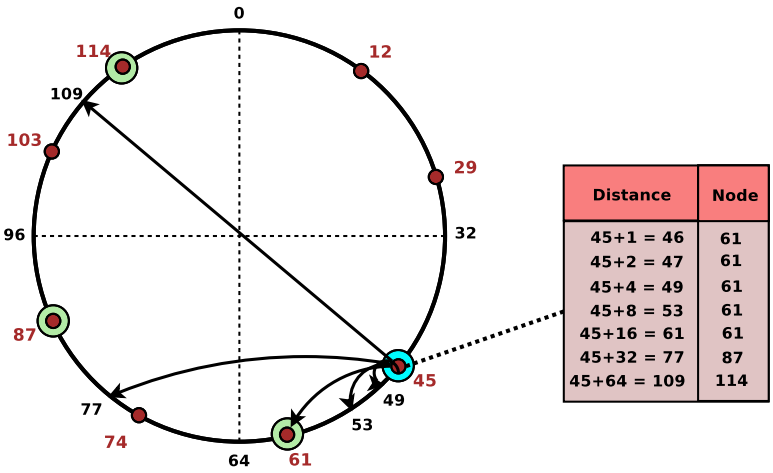


Fig. 4.1: Chord buckets and binary tree for 16 nodes.

When looking up the successor of any ID, a node can reach a contact that is located, at most, at half the distance from the target than himself. For example, in Figure 4.1, if node 45 needs to find the successor of ID 25, it will query node 114, which has a better knowledge of that part of the network. This in turn will contact node 12, which will contact node 29 which is the target. In general, look-up operations are completed in $O(log_2 N)$ steps.

The underlying distance metric used in this system is the ID subtraction. The distance between a node with ID A and a node with ID B is $\overline{AB} = B - A$ if $B > A$, and $\overline{AB} = N - (A - B)$ if $A > B$. For instance, in the

previous example, for $A = 45$ and $B = 61$, then $\overline{AB} = 61 - 45 = 15$ but $\overline{BA} = 128 - (61 - 45) = 112$ (because *one must turn clockwise*). Notice that this metric is not symmetric ($\overline{AB} \neq \overline{BA}$) and a node has a good knowledge of the nodes following it but not of the nodes preceding it (except for the immediate predecessor).

The Chord protocol also defines algorithms for node joins and leaves and a periodic maintenance routine to keep finger tables up to date, but we will not go into further details.

4.1.3.2. Pastry

Pastry was also one of the first DHT systems presented [91]. Like Chord, it also utilizes a circular ID space. The IDs are 128-bits digits of base 2^b (typically, with $b = 4$). In total, there are $N = 2^{128}$ usable IDs.

Nodes in Pastry keep a list of L close contacts: $\frac{L}{2}$ successors and $\frac{L}{2}$ predecessors. In addition, they keep a prefix-based routing table. Each entry in level i of a node's routing table points to an ID that shares the first i digits with the ID of the node. There are $log_{2^b}N$ levels and, at each level, there are $2^b - 1$ entries. When looking up an ID, a node will look for the entry that shares the larger number of bits with the target. If the routing tables are correct, the expected maximum number of steps required to reach any target is $log_{2^b}N$.

As we will see, Pastry's prefix-based routing is very similar to Kademlia's XOR-based bucket structure. However, the final routing step, when no longer common prefix can be found for the target, is performed using the list of (numerically) close contacts.

Authors of Kademlia say:

> *Of existing systems, Kademlia most resembles Pastry's first phase, which (though not described this way by the authors) successively finds nodes roughly half as far from the target ID by Kademlia's XOR metric. In a second phase, however, Pastry switches distance metrics to the numeric difference between IDs. It also uses the second, numeric difference metric in replication. Unfortunately, nodes close by the second metric can be quite far by the first, creating discontinuities at particular node ID values, reducing performance, and complicating attempts at formal analysis of worst-case behavior.*

4.2. Kademlia

Kademlia was presented in 2002 [79]. It is a successful DHT, used in real life applications like *Gnutella* [78], *Kad* [92] and *BitTorrent* [93].

Kademlia uses an ID space composed of all binary numbers between 0 and 2^{160}. Each node is identified by one of these numbers and, like in other

systems, values stored in the DHT are owned by the node whose ID is closer to the key associated to the value. The most important operation defined in a Kademlia network is the look-up operation, by which a node can discover and reach another node by its ID or the node where the value is stored by its associated key. The look-up operation is performed iteratively: in each step, nodes closer to the target are found.

A key concept for the look-up operation (and, thus, for Kademlia) is, naturally, the distance between nodes (what *closer* means). Kademlia computes the distance between nodes as the XOR (*Exclusive Or*) of their IDs; that is, if A and B are two members of a Kademlia network, then their distance is: $\overline{AB} = A \oplus B$. This means that, unlike in Chord, Kademlia distances are symmetric: $\overline{AB} = A \oplus B = B \oplus A = \overline{BA}$. This feature simplifies the analysis of the correctness of its algorithms and has several implications (as we will detail in the following paragraphs), one of the most important of them being that a node will learn about useful contacts passively when these other nodes perform look-ups. This is different to DHTs like Chord and allows Kademlia to avoid routine node discovery procedures.

In what relates to routing tables, each Kademlia node keeps a series of *buckets of K contacts*. Each bucket contains nodes within a certain distance range. In the simplest case, there is one bucket for each bit of node identifier and their associated distance ranges are $[2^i, 2^{i+1})$ for i in $[0, 160)$. This can be seen as a binary tree: the ID space is successively split in two, increasing the number of bits in common with the node ID by one. This is illustrated by the tree in Figure 4.2, which includes the Kademlia buckets for node 0101. Ovals contain nodes belonging to each bucket and, for each, the common prefix with node 0101 and the interval of distances to it are indicated.

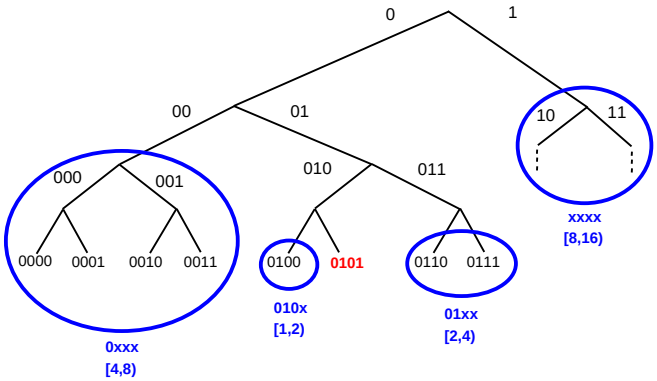


Fig. 4.2: Kademlia buckets and binary tree for 16 nodes.

The buckets can be made narrower in order to reduce look-up time at the cost of maintaining a larger routing table (using b bits instead of 1 for each bucket), thus having distance ranges of $[j \cdot T, (j+1) \cdot T)$ where

$T = 2^{160-b\cdot(i+1)}$ for j in $(0, 2^b)$ and i in $[0, \frac{160}{b})$. Each look-up iteration gets b bits closer to the target.

When looking for a target contained in a certain bucket, a node can use any of the contacts in the bucket as the next hop, because all of them are closer to the target that the node is. This is, again, a consequence of the use of the XOR metrics for distance calculations and, as before, it is different from the case of Chord, where only the contacts with the lowest IDs of a range can be used to search in the whole range (because they do not know about the nodes preceding them). Moreover, Kademlia is able to send requests to several contacts in parallel in order to gain redundancy and reduce latency in the reply.

Kademlia orders the nodes in the bucket in terms of recorded uptime. Older nodes are given preference for queries and new references are added to the contact lists only if there are not enough old nodes. This is due to their observation that, in existing P2P networks, nodes which stayed for a longer time in the past will tend to stay connected longer in the future than recently joined peers.

4.3. Broadcasting in DHTs

There have been many proposals to enhance DHT systems with broadcasting capabilities. Some of these works build new routing tables to create an overlay broadcasting tree [94]. This represents a burden of additional storage space and maintenance messages. We will not consider these approaches since the tables offered by Kademlia already contain enough information to build a tree that covers all nodes.

Some other systems are designed to work only with a particular DHT [95, 96]. Since they cannot be directly applied to Kademlia, they are not useful if this is the DHT of choice (e.g., in existing networks). There are however some general proposals that aim to be applicable to any DHT. We will describe them hereafter. In addition, we will also discuss recent work that does address Kademlia specifically.

4.3.1. Partition-based Broadcasting

One of the first proposals dealing with DHT broadcasting in a general way was presented by El-Ansary *et al.* [97]. They modelled look-ups as distributed k-ary searches where the ID space is divided in λ regions[1] and the query is forwarded to a contact in the region containing the searched key. This contact will in turn divide the remaining space in λ subregions and forward again to a closer contact. Broadcasting is achieved by just forwarding

[1]We use λ instead of the original K, not to confuse with Kademlia's K contacts per bucket.

data to a contact in every one of the λ parts and letting each contact do the same within its own region. In effect, the k-ary decision tree is transformed into a spanning tree. The algorithm is applied to Chord, where λ=2. The tree depth (number of forwarding steps) is $O(log_2 N)$.

The broadcasting tree produced by the k-ary search models is unbalanced. When a node tries to reach all of its contacts, it will send up to $O(log_2 N)$ messages while other nodes will send very few. If some nodes are often the source of the broadcasts, the first contacts in their trees will suffer from higher loads than other nodes. This is addressed by an new space partition algorithm, according to which, each node sends only two forwarding messages, effectively splitting the ID space into two parts each time (explicit 2-ary tree) [98]. This method achieves a balanced tree, with the same depth of $O(log_2 N)$. However, as we will see in Chapter 9, a balanced tree is more fragile when random network errors are present.

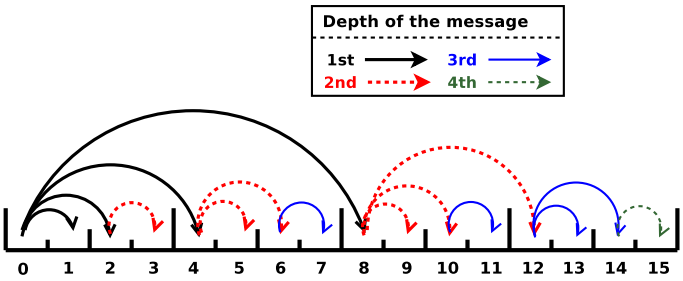
Figure 4.3 illustrates the unbalanced and balanced broadcast algorithms in a network of 16 nodes and originated at node 0. It is easy to see that, in the first case, there is one node (0) that sends 4 messages, one node (8) that sends 3 and the rest send fewer messages. In the second case, no node sends more than 2 messages. However, the depth of the broadcast tree is the same in both cases (4).

Different λ values can be considered [99]. With $\lambda > 2$, forwarding nodes will suffer from a somewhat higher load (they send more messages) but the tree depth will be reduced to $O(log_\lambda N)$, achieving an overall latency reduction. Such algorithm is regarded as general because it relies on DHT primitives to look up contacts in each partition. This is correct but the need to discover nodes before initiating a broadcast may increase latency. Original k-ary search just sends messages to already known contacts and that is what we aim to do with Kademlia too.

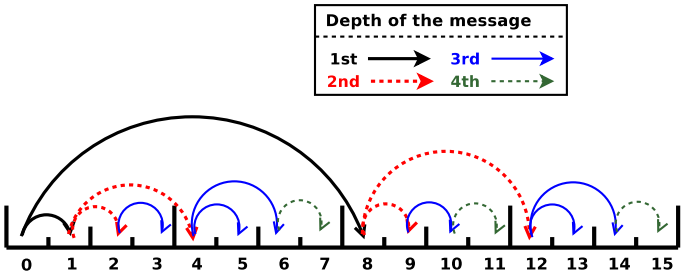
4.3.2. Prefix-based Broadcasting

Prefix-based DHTs are characterized by their routing tables: every node keeps an entry for each possible prefix common to its ID (considering one or several (b) bits for each step). Kademlia authors indicate that, although described in different terms, Pastry's prefix-based routing is very similar to that of Kademlia. Actually, Kademlia's buckets can be mapped to entries in a prefix-based routing table. When $b > 1$ bits are used, however, Pastry requires a second routing to discover the closest nodes from all sharing the same prefix but differing in the final digit.

The algorithm described by Wahlisch *et al.* is meant for this kind of prefix-based DHTs and, in particular, for Pastry [100]. The idea is to form a tree based on an increasing common prefix. At level L, a node forwards to all contacts whose ID has L bits in common with it. The same logic can



(a) Unbalanced k-ary broadcasting.



(b) Balanced partition broadcasting.

Fig. 4.3: Balanced and unbalanced broadcasting for a network of 16 nodes.

be applied to b bits symbols, instead of bits. The authors prove that, for complete prefix-based routing tables, the broadcast achieves 100 % coverage with exactly one message per node. Although this is only applied to Pastry, we will see that it can also be used with Kademlia, since, as indicated, it is possible to describe it as a prefix-based system.

4.3.3. Related Work on Kademlia Broadcasting

A detailed study of the broadcast operation in Kademlia networks will be provided in Chapter 9. Nevertheless, we would like to make a short note on related work on the topic, as follows.

When we started to look at the possibilities for broadcasting in Kademlia (with the objective of using it for our new distributed scheduling procedure), we did not find any specific work on this subject, only general purpose algorithms whose straightforward application to Kademlia was not completely clear. For this reason, we decided to undertake a more in-depth study of the problem and published our conclusions [101]. Later, Czirkos *et al.* presented a broadcasting algorithm for Kademlia [102]. Although developed independently, this algorithm turns out to be essentially the same as the one we

called *bucket-based* [101] (and which we will describe in Chapter 9). This is probably not surprising since this is, we believe, the most natural way to construct a broadcast tree in Kademlia. Moreover, like us, they also propose to apply redundancy to increase the reliability of the algorithm, profiting from Kademlia characteristics.

The contribution of Czirkos *et al.* is, in our view, valuable since they offer an analysis of the expected broadcast coverage given the network failure rate in the system, they consider—based on this analysis—different redundancy factors and they perform a study of the latency of the broadcast when non-uniform links are present. Nevertheless, their work does not offer a general overview of the possible approaches for broadcasting in Kademlia, comparing bucket- and partition-based protocols, with different division factors, and thoroughly studying all relevant metrics for each case. Likewise, other churn-fighting techniques, such as resubmissions, are not considered. Finally, they have simulated their algorithms while we have run ours in a real system with 1,000 independent nodes.

4.4. Churn and Failure Rate

The main problem with the algorithms described so far is that a failure in a node will cause a whole tree branch to miss the broadcast. This is especially likely in networks where nodes join and leave the system often (*churn*) and routing tables may contain stale information.

Out of the presented works, only two discuss churn [99, 102]. The first one proposes a simple ACK mechanism: each node waits for ACKs from the contacts it has sent a message to. Each contact will send an ACK only when it has verified that every node within its broadcast subtree has received the message (i.e., already replied with its own ACK). The problem is that the latency to notice that a message was lost may be quite high. The use of redundancy is also suggested [102]. This is also one of our proposed techniques, and discussed in depth later. Finally, flooding techniques may complement the core (structured) broadcasting mechanism to reach nodes that have missed a message [103].

In Chapter 9, we will discuss and evaluate the application of these and other techniques to broadcasting in Kademlia.

4.5. Evaluation Metrics

In order to assess the suitability of the different algorithms to the Kademlia DHT and to discuss their characteristics, we need to understand which are the most important metrics to take into account. Based on the literature (see, e.g., all the references given along this chapter), we conclude that the

most important factors are the following:

- *Coverage*: percentage of nodes receiving the broadcast message out of the total number of nodes in the system. This is obviously the most relevant metric for a broadcast protocol and in normal conditions should always be 100 %.
- *Messages to nodes ratio*: ratio between the number of sent messages and the number of nodes in the network. The higher this ratio is, the higher the load put on the nodes and on the network is. Ideally, this number should be 1.
- *Imbalance factor*: ratio between the maximum number of messages sent by any single node and the average number of sent messages per node. As discussed earlier, if the imbalance factor is high, some nodes may suffer from higher loads and impose more traffic on their network links. Again, the ideal imbalance factor is 1.
- *Tree depth*: Number of forwarding steps required to complete the broadcast. This is a measure of the latency of the broadcast, since the larger this value is, the longer it takes for the message to reach every node. Notice that this characterizes a given broadcast tree independently of the delay of individual messages.

We will take all these factors into consideration in our tests with the different protocols, in Chapter 9.

WRAP-UP: *We have presented the distributed hash tables and discussed their attractive scalability and robustness properties. DHTs are used in large distributed systems to provide look-up and overlay routing services, especially (though not only) in file storing and sharing applications. We are interested in DHTs because we will use one for a distributed data cache and for a task matching algorithm in our pilot architecture, discussed in Chapter 10.*

In order to better understand how DHT systems work, we have reviewed a couple of representative examples, Chord and Pastry, and looked into Kademlia—the DHT used in our architecture—in more detail. In addition, since our task scheduling procedure requires to broadcast messages to the members of the network, we have also discussed protocols to add support for this operation in DHTs. These include partition-based, prefix-based and bucket-based algorithms. Chapter 9 will study the use of these procedures with Kademlia, as well as the application of additional techniques to avoid message loss due to churn or node failures.

Part II

Architectures for Efficient Data Access

Chapter 5

Evaluation of Data Access and Task Binding

By describing the peculiarities and the challenges faced by data-intensive workflows in scientific grids, the previous chapters have set the context of our work. They have discussed related work and the current state of the art. Now, this chapter prepares the reader for the first set of our contribution work, by providing the necessary background information and our specific motivations. Subsequently, it introduces our ideas for enhancement and the lines of research that will be further developed in succeeding chapters.

Since the object of our study is the improved execution of *data-intensive* workflows, it is obvious that the data to be processed/produced by those workflows must be a fundamental factor in our analysis. Indeed, the task of providing computational jobs with efficient access to large amounts of data is one of the main challenges faced by many scientific workflows, specially in the grid, where resources are diverse and geographically disperse. Providing for adequate data access for jobs implies being informed about the resources storing required files and incorporating that knowledge into the scheduling process, as well as trying to make sure that storage systems are capable of serving the data efficiently. Both aspects are discussed in Section 5.1.

Section 3.1 introduced the late-binding architectures and described how they are now the predominant scheduling systems for all the major VOs in WLCG. They offer clear advantages, such as robustness against resource failure, interface homogenization and the ability to centrally and dynamically assign priorities to the VO's workflows. In Section 5.2, we study the performance of executions following one approach and the other. In order to do this, we develop an enhanced version of an existing analytical model.

Finally, Section 5.3 introduces the concept of *micro-scheduling*; i.e., allocating tasks to particular nodes rather than globally to resource centres. The micro-scheduling techniques are actually required for our proposed data cache, discussed in Section 5.1.3 and made possible only by using a late-

binding overlay.

5.1. Data Access

5.1.1. Collocating Jobs and Data

As seen in Section 3.3, data-intensive job management systems consider data location to be an important parameter for job scheduling. This is widely accepted by the community, generally because of the time wasted transferring input data required by jobs. Chapter 6 analyzes the problem in more detail, including other issues caused by extensive grid file replication, and provides some experimental tests on data staging. Furthermore, we are not only interested in the scheduling at the site level but also within sites (WN level), in order to increase data access throughput for applications and to protect the storage systems at the sites. This is further discussed in the following sections.

5.1.2. Accessing Storage Elements Data

Even if WLCG storage elements are designed to store petabytes of data and deal with hundreds of clients, they are not infinitely scalable and may suffer from performance problems if they are exposed to a high number of very active clients. This was discussed in Section 3.2.1. It is true that the situation is better now that it was at the time that we started our work, in particular at CERN's Tier-0 (which was the use case that originally motivated our research) but, still today, SEs are not immune to congestion and performance degradation.

The response time and reliability of grid SEs cannot be described by a constant function. Depending on the load they are put under and the access patterns of the clients accessing the storage, the perceived performance of the SE might be different. Since SEs are fairly complex pieces of software and the technologies used for massive storage management are certainly evolving (and so are the load levels the SEs must deal with), it is out of the scope of our work to characterize the effects that stressful activities cause on such systems. For us, it suffices to understand that these effects exist (as experience has shown) and focus instead on studying how we can protect ourselves from these problems and, if possible, avoid causing them.

In addition, as seen in Section 3.4, current trends in scientific grids, such as the WLCG, show that VO workflows are nowadays subject to more heterogeneous resource types and data access patterns. VOs are currently often capable of running in the cloud, with expensive or suboptimal (low bandwidth, high latency) network links to the VO's mass storage systems. Also, they are prepared to run on resources with no local storage at all

(Tier-3 sites, volunteer computers), being obliged therefore to access remote storage (through WAN), with expanded latencies. We would like to minimize WAN data accesses when possible.

5.1.3. Pilots Data Cache

We propose to implement a data cache in the nodes executing VO's workflows, so that SEs can be relieved of some pressure and jobs can gain some CPU time by being able to access input files locally. In addition, this possibility may be even more interesting for cloud resources or sites without mass storage if the cache helps to avoid some very costly remote data transfers.

Jobs running on the grid consume a certain amount of disk space. This is unavoidable, since jobs need to, at least, store their own code, configuration files and produced logs. However, it is very frequent that, in addition, jobs download (stage-in) some input files from storage services or that produce data files as output of their execution. These files are usually of larger size (several GB) and require the sites to provision worker nodes with sufficient space for the expected number of jobs running concurrently on them. E.g., the CMS VO usually asks sites to guarantee at least 10 GB of disk per job slot on a WN.

The idea behind our data cache is to take advantage of the space that the WNs offer to the real jobs they are executing. Given that pilots create an overlay on the existing resources, they can take charge of managing a cache so that locally stored files can be reused by different tasks and transfers from/to mass storage are reduced as much as possible. As we will see, an effective usage of this cache requires also the application of *m*icro-scheduling techniques. We have proposed a new pilot-based late-binding overlay system, implementing the cited data cache, called the *T*ask Queue architecture. Chapters 7 and 10 describe and evaluate this architecture and the implemented pilot-based data cache.

5.2. Early-binding vs Late-binding

This section presents an analytical study of the performance of early- and late-binding scheduling approaches, in terms of latencies and feasible turnaround times. We discuss a model that evaluates these subjects and, in addition, serves as the foundation for a further discussion on computational task granularity and the importance of matching and scheduling delays, presented in Section 8.2.

5.2.1. Modelling Early- and Late-binding Approaches

As a starting point, we will discuss the model proposed by Moscicki *et al.*, which tries to characterize early- and late-binding approaches in the processing of known and finite user workloads on the grid [104].

5.2.1.1. General Workflow Execution Model

In the initial model, the workload can be divided into arbitrarily small tasks. Grid jobs are submitted using traditional means and start to run after some initial delay T_i^s, which is different for each one[1]. The jobs then run one or several workflow tasks for some time T_i^φ until they finish. Notice that the initial model assumes that the maximum runtime per slot, as enforced by the local batch system, is long enough not to be taken into account. In general terms, there is a time T_i^e, after the last computed task, in which a slot is either idle or computing for a different workflow but anyway not productive for the workflow at hand. This model is illustrated by Figure 5.1.

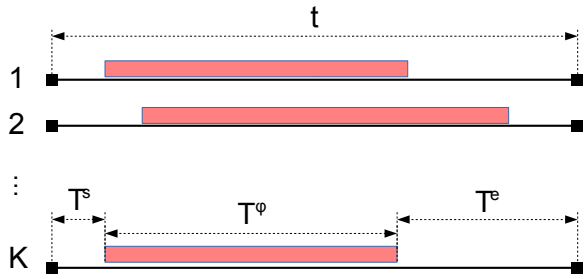


Fig. 5.1: General model for task execution on grid slots.

Given the previous definitions, the formula for the completed workload as function of time is:

$$W(t) = \sum_K W_i(t) = \sum_K \int_{T_i^s}^{t-T_i^e} \varphi_i(s)ds = \sum_K \int_0^{T_i^\varphi} \varphi_i(s - T_i^s)ds \qquad (5.1)$$

where:

- $W(t)$ is the total work completed in t seconds (using all processors).
- $W_i(t)$ is the work completed in t seconds at processor i.

[1]The exact delay of an individual job depends on factors such as the latency of the submission machinery, the queue length at the computing resources and the initialization time of the task once on the computing slot.

- K are the available parallel computing slots.
- $\varphi_i(t)$ is the computing speed of processor i as a function of time
- T_i^s is the initial delay (scheduling, queuing, initialization) of the task at slot i.
- T_i^e is the time elapsed between the end of the task at slot i and t.
- T_i^φ is the time the node is computing, such that: $T_i^\varphi = t - T_i^s - T_i^e$.

To simplify, we will consider all processors to have the same constant processing speed: $\varphi_i(t) = \varphi$. Thus, for a single slot:

$$W_i(t) = \varphi(t - T_i^s - T_i^e) \tag{5.2}$$

So, if L is the time taken for the workflow to complete, the total computed work during L seconds, using K slots will be:

$$W = \sum_K \varphi(L - T_i^s - T_i^e) = \varphi(LK - \sum_K (T_i^s + T_i^e)) \tag{5.3}$$

And so, if we know the workload W that we need to compute, we can calculate the duration of our workflow with Equation 5.4:

$$L = \frac{W}{\varphi K} + \sum_K \frac{T_i^s + T_i^e}{K} \tag{5.4}$$

5.2.1.2. Early-binding

In early-binding, the workload to complete is divided in K tasks (each one of duration $\frac{W}{\varphi K}$), so that exactly one is sent to each available processor, in order to complete the work as quickly as possible. When the last task is finished, the workflow has been completed. This is illustrated by Figure 5.2.

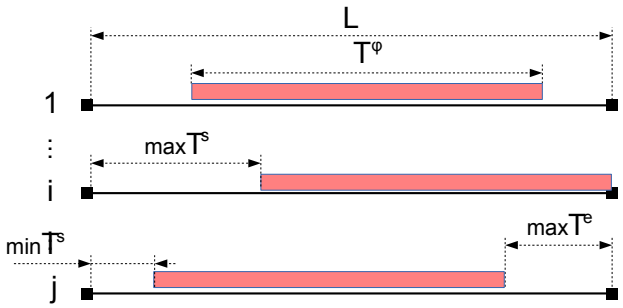


Fig. 5.2: Early-binding model for workflow execution on the grid.

As said, in this model, $T_i^{\varphi} = T^{\varphi}$ (constant), so the sum $T_i^s + T_i^e$ is also constant for every i. Furthermore, this sum must be equal to $max(T_i^s)$, since, for the pilot with the maximum T_i^s value (last task), $T_i^e = 0$. Thus:

$$T_i^s + T_i^e = max(T_i^s) \tag{5.5}$$

which is constant. So, we reach Equation 5.6:

$$L = \frac{W}{\varphi K} + max(T_i^s) \tag{5.6}$$

5.2.1.3. Late-binding

In late-binding, new tasks can be scheduled to a pilot as long as we still have time left and the workflow has not finished yet. Since, in the initial model, tasks can be as small as desired and there is no idle time between tasks, we can fill the slot with useful work completely, exactly until the end of the workflow.

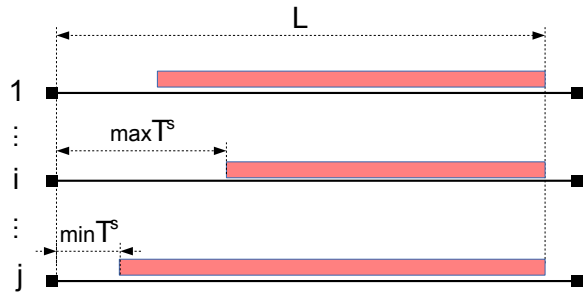


Fig. 5.3: Late-binding model for workflow execution on the grid.

As a result of this, $T_i^e = 0$ for all pilots and T_i^{φ} is not constant but dependant on T_i^s. We can see this in Figure 5.3, where a shorter L is achieved due to the higher contribution of all nodes except for the one with maximum T^s. Here, we have:

$$\sum_K \frac{T_i^s + T_i^e}{K} = \sum_K \frac{T_i^s}{K} \tag{5.7}$$

where T_i^s can be considered random and the right hand side of the equation is the average T_i^s value. Replacing in Equation 5.4, we reach:

$$L = \frac{W}{\varphi K} + avg(T_i^s) \tag{5.8}$$

L is smaller in late-binding because the slot can be filled better. In particular, wasted time is $avg(T_i^s)$, while in early-binding it is $max(T_i^s)$.

This difference can be significant if the distribution of start delays has a long tail, as it is usually the case with real grid workflows [104].

5.2.1.4. Discrete Model

Up to now, we have discussed the grid execution model for continuous user workflows. But, in the real world, a workload cannot be divided into arbitrarily small bits. Now, we enhance this model by considering discrete tasks and inter-tasks delays. This is also mentioned by Moscicki *et al.* but only some simulations are provided. In contrast, we derive an analytical expression, which will be used in Chapter 8 for further study.

In the following, we will discuss the late-binding architecture, but most of the reasoning can be equally applied to the early-binding case. In both, we consider an initial delay T_i^s (product of traditional submission and resource queuing), the execution of a discrete task (of duration T^t) and, between successive tasks, a match-making and start-up delay (T^m). The main difference between the two models resides in the value of T^m. In general, we will find that this is higher for the early-binding case (see Section 5.2.2 for a more in-depth discussion on this).

Figure 5.4 shows the execution of a workflow using a late-binding architecture. Task i is the one marking the end of the workflow. For other slots, there is not enough time for even one more task, so there is a non-null T_i^e. For one slot and N_i being the number of tasks run on slot i, we have:

$$L = T_i^s + N_i(T^m + T^t) + T_i^e \tag{5.9}$$

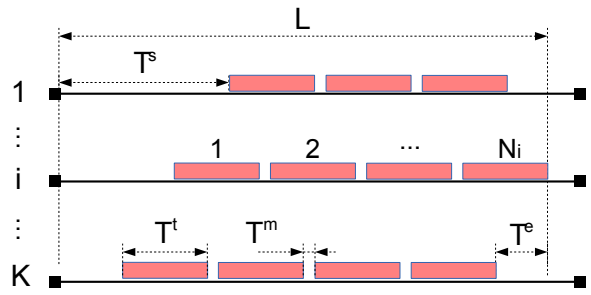


Fig. 5.4: Discrete model for workflow execution on the grid.

Since more tasks are scheduled while there is time for them, we will have $T_i^e \leq T^m + T^t$ and $N_i = floor(\frac{L-T_i^s}{T^m+T^t})$. For K slots:

$$KL = \sum_K (T_i^s + T_i^e) + \sum_K N_i(T^m + T^t) \tag{5.10}$$

However, we also know that (considering all the K slots):

$$W = \varphi \sum_{K} N_i T^t \Longrightarrow \sum_{K} N_i = \frac{W}{\varphi T^t} \tag{5.11}$$

This means that:

$$KL = \sum_{K} (T_i^s + T_i^e) + \frac{W}{\varphi T^t}(T^m + T^t) \tag{5.12}$$

And:

$$L = avg(T_i^s) + avg(T_i^e) + \frac{W}{\varphi K} + \frac{W}{\varphi K}\frac{T^m}{T^t} \tag{5.13}$$

If we introduce a new constant $C = \frac{W}{\varphi K}$, then we have Equation 5.14:

$$L = avg(T_i^s) + avg(T_i^e) + C + C\frac{T^m}{T^t} \tag{5.14}$$

Which is similar to the previous model (Equation 5.8) but with the addition of two new terms $avg(T_i^e)$ and $C\frac{T^m}{T^t}$. The first one represents the inability to always fill the slot completely with tasks of finite duration. The second one represents the time that is lost in match-making and task download and start-up (proportional to the relation $\frac{T^m}{T^t}$).

5.2.2. Workload Throughput Considerations

In the previous sections, we have focused on the completion time of a finite workflow. This is especially interesting for individual scientists and relatively brief workloads. However, focusing on VOs as a whole, we find that these usually run jobs over long period of times and they are probably more interested in throughput than in latency. Rather than asking how long it takes to complete an individual workflow, they ask how much work can be done with a given number of resources for a given time. That is why we now apply our previous model to this situation and see how the early- and late-binding models behave in this case[2].

5.2.2.1. Early-binding

The early-binding model is illustrated by the Figure 5.5, which, for simplicity, shows only tasks that belong to the workflows of interest. We can think of it this way: at any time, the VO has K available processors; it is not really important for our purposes whether these are always the same or not (in general they are not). We see that, in this case, the resources run tasks continuously. The only idle time is the inter-job delay T^m, between

[2]The contents of this section are not discussed by Moscicki *et al.*

the end of a batch job and the start of the next one. The T^m value is the key to the achievable throughput and several factors contribute to it: submission delay, queuing time, batch system job initialization time and VO task initialization.

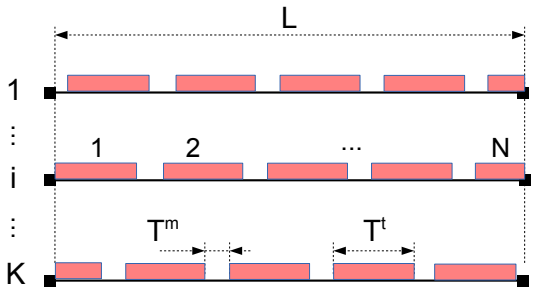


Fig. 5.5: Workload throughput model on early-binding approach.

The first two factors are not trivial to characterize. If jobs are sent on bursts, only the first ones to arrive at the site will exhibit a submission delay. The queuing delay will depend on competing workflows and site priorities. However, in general there will always be inefficiencies: some jobs are sent just when needed (non-null submission delay) and, at occasions, they may linger on sites queues longer than expected (i.e., at a particular moment, fewer than K processors are active). All in all, we can just say that all these delays are included in certain T^m value, which, in principle, should not be high for VOs with available resources, but it is certainly not negligible.

5.2.2.2. Late-binding

The late-binding approach is illustrated by Figure 5.6, which shows how a resource runs batch jobs and each one of those, in turn, runs M tasks (which are assigned at runtime). This time, we find two different idle times: T^{m_1} and T^{m_2}. The first one is an *inter-job* delay and the second is an *inter-task* delay (between tasks run on the same batch job). It is clear that if $T^{m_1} = T^m$ (the delay in the early-binding model) and $M > 1$, then the late-binding model will behave better than in early-binding if $T^{m_2} < T^m$.

We cannot provide firm estimates for all those values. However, it is clear that if a VO has resources dedicated to it and jobs are sent beforehand to the batch system queues, then T^m may be lower than T^{m_2} (task request, matching and download to the pilot). If this is the case, then the early-binding model may provide a higher throughput.

On the other hand, in the general case, VOs compete for resources and the flow of jobs is not continuous, so with non-null submission delays, we believe it realistic to assume that $T^{m_2} < T^m$. In addition, VOs using the

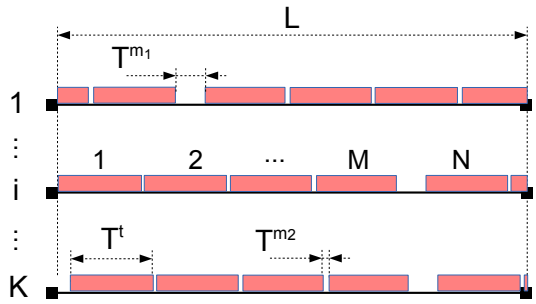


Fig. 5.6: Workload throughput model on late-binding approach.

late-binding model may keep a pressure of pilots on sites[3], so resources are always available for newly created tasks. Combining this with the possibility to reorganize priorities on the central task queue, we find that $T^{m_1} < T^m$ also and, thus, the throughput achieved with the late-binding approach will be higher than that of the early-binding one.

Finally, M is the number of different tasks that a pilot can run sequentially. It depends on the maximum allowed batch system runtime and the length of individual tasks. Ideally, VOs could submit really long tasks, making $M = 1$. In practice, site policies vary and VOs may need to play safe and use shorter lengths to avoid exceeding the most restrictive time limits. Furthermore, task length is often dictated by external factors (manageable size of input/output data, experiment constraints, etc.) and is considerably lower than allowed job times. Therefore, it is common that $M >> 1$.

5.2.2.3. Summary of Throughput Considerations

The achievable workload throughput by centrally managed VO workflows depends on the amount of time that the assigned resources are idle during scheduling, queuing and initialization phases. This is more difficult to characterize than in the latency model. We can speculate that, in general, late-binding architectures behave somewhat better than early-binding ones, however, this may not always be the case. What we know for sure is that, in order to improve results, the T^m value should be minimized. For late-binding architecture, this implies keeping a constant pressure of pilots on the sites and making all efforts to reduce the task matching delays.

Finally, please recall that even in the cases where work throughput or latency are not improved, the late-binding approach offers other advantages

[3]This approach will cause a fraction of the pilots to run empty, which means that the VO will be consuming resources for no real work, which is clearly inefficient. VOs must balance how many pilots they need to maintain on sites, depending on resource consumption policies (or prices), amount of work to be done, etc.

(robustness, homogeneity, dynamic priorities), which make VOs regard it as a superior option than the traditional scheduling.

5.3. Intelligent Micro-Scheduling

Large grid scheduling traditionally regards sites as homogeneous sets of resources. At most, a site offers a few different queues with varying job length limits or different memory capacities but the number of resource groups has to be necessarily very limited to make it manageable by both users and site administrators [24]. However, in late-binding systems, the central server of a VO assigns jobs directly to individual pilots, so that it can potentially consider the particular characteristics of each node (rather than those of the site or queue). We call this possibility *micro-scheduling*.

The primary motivator for our micro-scheduling work is that it is essential for an adequate usage of the pilot-based data cache discussed in Section 5.1.2. An effective data cache requires a high *hit ratio* and this is only possible if computing tasks are scheduled to the appropriate execution nodes—those holding the data they depend upon. Micro-scheduling (selecting the exact pilot a task will run on) is, thus, necessary for this goal. Moreover, even if not implemented at the moment, if a distributed file system was available on the WNs and file-location information was made available to our pilots, they could match input data requirements from tasks to stored files, just as they do with cached files. This would represent a huge improvement in terms of network requirements for the site. An evident candidate would be HDFS, employed as SE in WLCG but disregarding its data locality information (refer to Section 3.3.2 for details).

Nevertheless, beyond file location matching, micro-scheduling is a powerful technique, usable in other interesting ways. A fine-grained task matching would make it possible to select work for a pilot based on its precise characteristics, such as memory, processing power, disk space, attached instruments, software licenses, etc. More particularly, information about the jobs running in a pilot's node could be used to balance the number of I/O-bound and CPU-intensive tasks so that the WN was used more efficiently. Or, now that the WLCG VOs are in transition from single-core jobs to single-node but multi-core tasks, with the aim of improving the efficiency of disk and memory management[4], the pilots could also advertise the number of available CPUs at a node to pull tasks of different sizes to the WN [105].

Notice also that even if traditionally scientific grid environments have been fairly homogeneous environments (at least, in relation to operative

[4]This presents however a difficult problem in how nodes are filled. Imagine, e.g., that a site is full and all the WNs are running single-core tasks but we have a few 8-core tasks to run, we will need to drain an equivalent number of slots and while the last of the single-core tasks finishes, all the other cores are idle, so resources are being wasted.

system and available middleware), VOs are now exploring the possibilities to run in other environments such as the cloud or in volunteer computing platforms. In such cases or in future ones, the range of heterogeneous computational environments may increase and being able to match tasks and resources with fine granularity may become more and more necessary for an efficient usage of the resources.

Chapters 7 and 10 discuss the attainment and application of micro-scheduling in our proposed Task Queue late-binding overlay.

WRAP-UP: *This chapter introduces some of the basic ideas that we have proposed for our Task Queue architecture. The first one of these is the implementation of a data cache on the local disks of the nodes where pilot jobs are run. This cache will be used to improve data access performance and protect storage systems. The second one is the use of a fine-grained task assignment procedure, so that particular nodes (rather than sites) are selected for task execution. The immediate application of this capability is the consideration of data location information for scheduling but other uses are possible.*

In addition, the chapter discusses a model to compare the performance of early- and late-binding approaches for workflow execution. The late-binding model seems clearly superior for finite (individual) workflows while things are more complex when looking at VO's global throughput (though late-binding is in any case preferred for other reasons). This performance model will be considered again in Chapter 8.

Chapter 6

Data-location Aware Scheduling

The key role played by the *data* in the execution of many scientific workflows, and specifically of those of WLCG, has been stressed several times already. At this point, it seems clear that *where* the data is stored and *how* it is accessed are essential factors for the allocation of computational tasks. In the case of the grid, scheduling systems must consider which sites hold the required data before submitting grid jobs.

This chapter presents our initial work on data-location aware scheduling. It offers concrete evidence of the importance of this information and presents a scheduler modified to use it in a more flexible way than WLCG's WMS. However, the problem of balancing the best computing resources and those storing the needed data is fairly complex. More sophisticated solutions would be required to tackle it in a more ambitious way. Hints on how these systems could look like are provided.

As indicated, this is an early work. But, even if a traditional scheduling approach was used (while we nowadays favour late-binding) and the work explicitly deals with inter-site—macro—scheduling (and we have later turned our focus into micro-scheduling), we believe that the described issues and the learnt lessons can also be of application to other task allocation contexts, such as the mentioned ones (late-binding, intra-site).

6.1. Data Location Awareness

If grid jobs are scheduled to sites whose SE does not hold their required data, the jobs need to bring the data to either that SE or the computing node itself. In order to assess the impact of the transfer of data relative to the total job turnaround time, we performed some transfer measurements between two different grid sites—a nearby and a distant one—to the third,

local, site. We use the adjectives *nearby* and *distant*, in the sense that the achieved data throughput is higher in the first case than in the second one. This is typically due to a better network link connecting the sites (e.g., same national network infrastructure or dedicated link), a smaller latency value, or to more appropriate tuning of configuration parameters (e.g., properly dimensioned TCP (*Transmission Control Protocol*) buffers). Transfers from the local SE to the WN were also conducted.

In every case the replicated files had a size of 2.5 GB. These tests were firstly performed in 2008. The results are summarized in Figure 6.1, which shows the delays obtained for a series of transfer iterations. What we find here is, firstly, that transfer to the WN take much longer than those to the local SE; except for the replication from the local SE to a local WN (both sitting within the same local network). This is to be expected, since storage systems are designed and tuned precisely to transfer data, while WNs are not optimized for that.

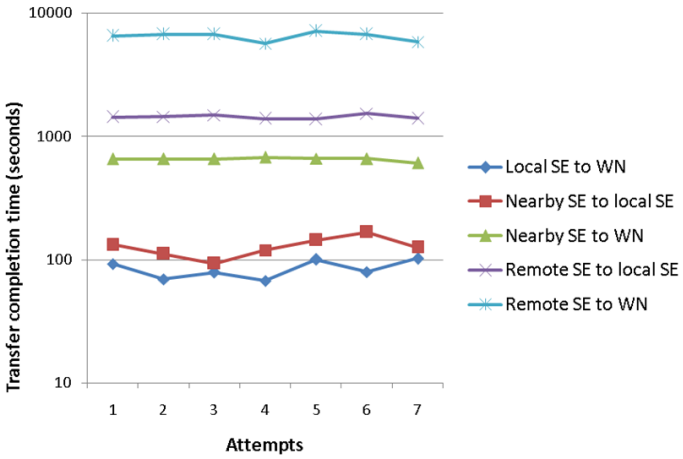


Fig. 6.1: Transfer completion times for various source and destination types.

The times in Figure 6.1 were measured for the copy of a 2.5 GB file. Considering the CMS case as an example, we can translate this into 41.6 minutes of processing time[1], though, some inefficiency and startup time should be factored in. It is clear that the delays incurred by the transfers to the local SE, as shown by the plot, are significant: more than 2 minutes for the nearby SE (19 MB/s), which might be tolerable, and almost 24 minutes for the distant site (1.7 MB/s), which does not seem acceptable.

Since these measurements show a particular case, let us mention here that recent monitoring reports[2] show values of 5 MB/s for average WAN,

[1] CMS reports processing speeds around 1 MB/s.

[2] `http://dashb-wlcg-transfers-dev.cern.ch/ui/#date.from=201408120940&date.interval=0&date.to=201408121040&tab=history_plots&vo=%28cms%29`

SE-to-SE, transfers in CMS[3]. If we take this number for the transfer rate, the delay for the copy from a remote SE to the local one would be of 8.3 minutes (without considering some overheads for authentication, set-up and ending).

No matter what exact transfer rate we use, it is clear that the delay induced by the replication of the input data is not negligible in comparison to the job processing time. This is the most obvious problem with on-demand (triggered by jobs) data replication, however it is not the only one, as we will see, as follows.

6.1.1. Data Replication and Management

Uncontrolled data replication, as demanded by running jobs, has other undesired consequences apart from wasted CPU time. One of them is the competition for storage and network resources, which should be devoted to other tasks (e.g., distribution of fresh detector data). Another one is the possibility that the existing storage space is exhausted, causing the abortion of the jobs. Depending on the available storage capacity and VO policies, these issues may end up being more problematic than the transfer delays.

Some of the works discussed in Section 3.3 showed that only by paying attention to both data transfer delay and job queueing/execution times, optimal scheduling—i.e., minimal job turnaround time—is achievable. It is sometimes more convenient for a job to wait for the replication of the input data than to wait for a busy computing resource to offer a free slot where to run. But we also indicated that estimating all these times is not always feasible (definitely, not easy) on real production infrastructures. Notice however that even if we could optimize job efficiency successfully, we would in general not minimize data replication, what, as just indicated, would cause other problems related with storage systems themselves and overall data transfers performance. This may not penalize the efficiency of a particular job but it will in general affect the global efficiency of a VO. We therefore believe that any algorithm assessing the convenience of the movement of data should apply a factor to offset the effects that such data movement may bring in addition to the unused CPU time cost.

The approach of EMI's WMS was to always send jobs to where data is, regardless of the situation of the computing resources [24]. The submitter might indicate the required files and the WMS would make sure that only

[3]We have also seen more optimistic and more pessimistic reports

- `https://indico.fnal.gov/getFile.py/access?contribId=14&sessionId=14&resId=3&materialId=slides&confId=5610`
- `https://indico.egi.eu/indico/materialDisplay.py?contribId=245&sessionId=53&materialId=slides&confId=1417`

We believe however that 5 MB/s from a random WN is already a bit optimistic.

sites holding these were considered for execution. In this way, data movements are minimized. VOs would arrange data location by other means, for example by using their own DPS's. This simple approach seems useful enough for HEP experiments but it is suboptimal in certain cases where jobs may land on congested or inaccessible sites. For instance, if only one site owns a replica of the data the user is interested in and this site is down or has many jobs waiting in the queue, then the user cannot run her analysis (or it will take a really long time). In this case, though, a manual copy of the data can be made (or requested to the central operations teams).

6.2. The GridWay Meta-scheduler

GridWay is a metascheduler that uses Globus core services to offer higher-level functionality to applications and users, thus simplifying the use of the grid [10]. GridWay offers a batch system-like command line interface for users to submit, kill migrate, monitor and synchronize jobs, as well as to watch information about available resources. GridWay was designed with a modular architecture, in which several components may be loaded as plugins. There are plugins for file transferring, job submission and information retrieval. Thanks to these, GridWay is able to submit jobs to both web services-based Globus resources and WLCG resources.

GridWay's scheduling algorithm makes use of configurable system policies and per-job user conditions and indications. The latter are given in a job template via requirement and rank expressions, similar to those of EMI's WMS, but with the limitation that, in GridWay's job template, there is no way a user can indicate any data needs.

6.2.1. Data-location Aware GridWay

To solve this shortcoming, a modified GridWay prototype was developed, which defined several new functions for both the requirement and the rank expressions. By using these functions, the presence and the size of needed data could be taken into account in the evaluation of the computing resources. Each of the added functions received the LFN (*Logical File Name*) of the required file. The new functions were the following:

- `CLOSE_DATA(LFN)`: Requirement expression. For each CE, it was evaluated as true only if specified data was held by a local SE and as false otherwise.
- `HAS_CLOSE_DATA(LFN)`: Rank expression. For each CE, it was evaluated as 1 if specified data was held by a local SE and as 0 otherwise.
- `SIZE_CLOSE_DATA(LFN)`: Rank expression. For each CE, evaluated as the size of the specified data if held by a local SE and as 0 otherwise.

This approach was more flexible than that of the EMI WMS, which only allows for data needs to form part of the requirement expression; i.e., either jobs are always sent to where data is or data location is not considered at all. On the contrary, the modified GridWay allowed for the inclusion of data functions in the rank expression as well. This means that the user (or VO) was free to decide how to weight factors like data presence, processor speed, number of queued jobs, etc. This made it also possible to include a custom factor to offset excessive data movements, as described earlier.

The information regarding the location of data was obtained by querying a grid data catalogue. In particular, our base implementation supported queries to the DLI (*Data Location Interface*) [106]. Since the DLI did not offer data size information, the `SIZE_CLOSE_DATA` function could not be used with it. However, other catalogue types, like the LFC (*LCG File Catalog*) were tested (and used in the evaluation of Section 6.3) and could have been easily added to a GridWay distribution [107].

In order to avoid excessive catalogue queries, data location information was cached. The entries in the cache contained a timestamp so that old entries were discarded and queried again. GridWay was also modified to inform the job about the list of all the files that were requested in the rank expression but were not located in the finally selected destination. In this way, the job was made aware that it might need to replicate those files.

As a summary of what has been described up to now, the diagram in Figure 6.2 schematically represents the process triggered when GridWay's job template parser encountered a `HAS_CLOSE_DATA` function in the rank expression. This process was followed for each possible destination host. Firstly, the argument of the function was saved in the list of requested files for the job. Next, if the required information was already in the cache and it was not too old, it was used for the evaluation. Otherwise, the catalogue was queried and the response cached. If any of the close (local) SEs associated to the candidate destination held the specified data, then a True value was returned. Otherwise, False was given back.

6.3. Evaluation

Several experiments were conducted in order to evaluate the functionality and performance of our implementation. All were done using resources of the WLCG infrastructure and standard EMI middleware.

6.3.1. Delay Introduced by the Catalogue Queries

Firstly, we measured the delay that the catalogue queries introduced in the match-making process. Figure 6.3 shows the match-making time for a series of jobs. In this example two required input files were queried in the first

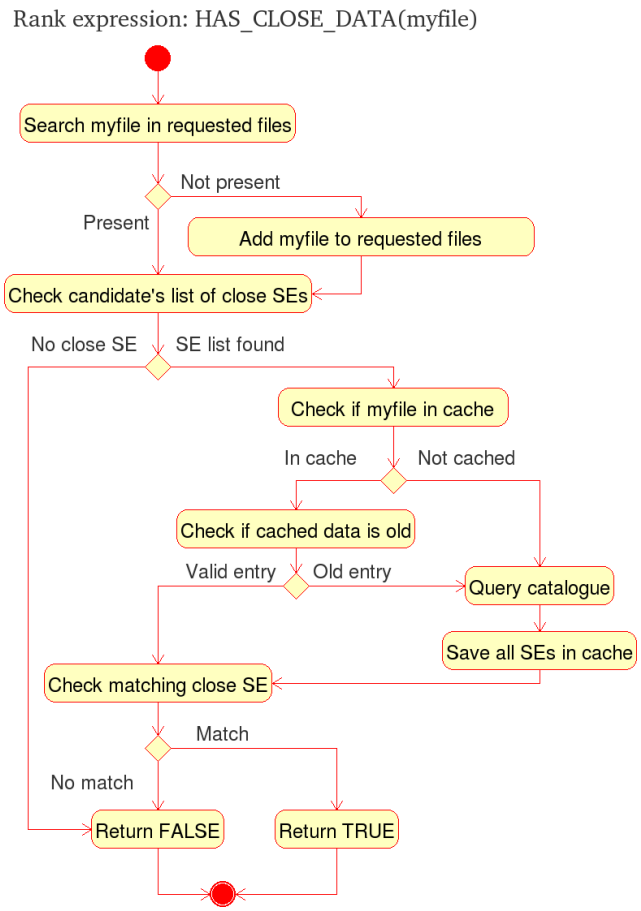


Fig. 6.2: Activity diagram for the resolution of the `HAS_CLOSE_DATA` function in GridWay.

attempt. On the fourth attempt, three additional files were included in the ranking and were also queried for. In the rest of the cases, the requested files were already in the cache, what greatly reduces the overall match-making time. Moreover, Figure 6.4 shows that the match-making time was, in any case, negligible if compared to the time it took the job to start running at the selected resource.

6.3.2. Application of Different Scheduling Policies

We also checked that our prototype was able to implement different policies. We considered three different sites: A, B, C. The first two had faster processors (our test job took around 500 seconds to run there) while the last one was slower (around 675 seconds per run) but this one held the input data the job demanded. We repeated the test for sizes of data ranging

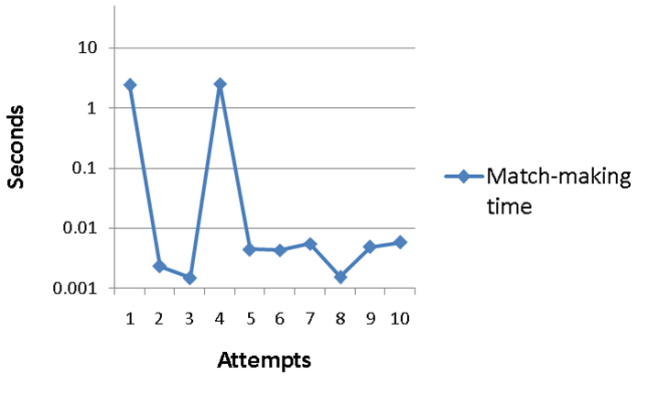


Fig. 6.3: Match-making delay in GridWay when catalog queries are added.

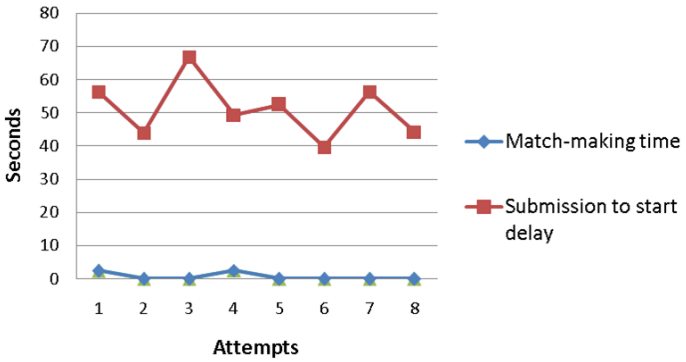


Fig. 6.4: Match-making delay compared to job submission delay.

from 3 to 405 MB (with the same processing speed) and for four different policies described in Table 6.1. Some of these policies use the `*CLOSE_DATA` functions, which get a positive value for sites holding required input data and 0 otherwise.

In the table, (d) can be considered the general expression, where `CPU_MHZ` indicates the speed of the processor, Kt is a conversion factor and Kp is a penalty factor added to offset the general negative impact of data transfers. Kt is used to estimate transfer delays depending on file sizes (thus, it should be inversely proportional to the expected transfer rate), but also to make this delay comparable to the `CPU_MHZ` value (i.e., how much shorter the job will run given an improvement in the CPU speed). Certainly, finding a perfect formula is impossible since one can never be sure how long the processing or the transferring of certain amount of data will take. However, at least there is a means for some balancing to be done and experience may teach the most adequate values for the average case. In this example, our tests revealed that using $Kt = 0.0075$ was appropriate.

Table 6.1: Policies.

Policy	Requirements	Rank
(a)	none	`CPU_MHZ`
(b)	`CLOSE_DATA(myfile)`	`CPU_MHZ`
(c)	none	`CPU_MHZ + Kt * SIZE_CLOSE_DATA(myfile)`
(d)	none	`CPU_MHZ + Kp * Kt * SIZE_CLOSE_DATA(myfile)`

Every rank expression can be seen as a particularizations of (d), with different values of Kp. For (a), we have $Kp = 0$, which means that destinations are selected based on CPU only. For (b), $Kp = \infty$, since the `CLOSE_DATA` requirement forces jobs to be submitted to the site storing the data (C, in this case). For (c), $Kp = 1$, which means that job processing and data transferring times are balanced, but no penalty is added to minimize data replications. Finally, for (d), we used $Kp = 2$, to penalize transfers by doubling the score of the data factor for sites holding that data.

The results of the test are given in Figures 6.5 and 6.6, which show the selected site for each case and the total turnaround time, respectively. We can see how policy (a), which does not take data location into account, always sends to sites A or B while policy (b) always sends to site C. However, policies (c) and (d) select the first two sites for small data sizes and C for bigger files.

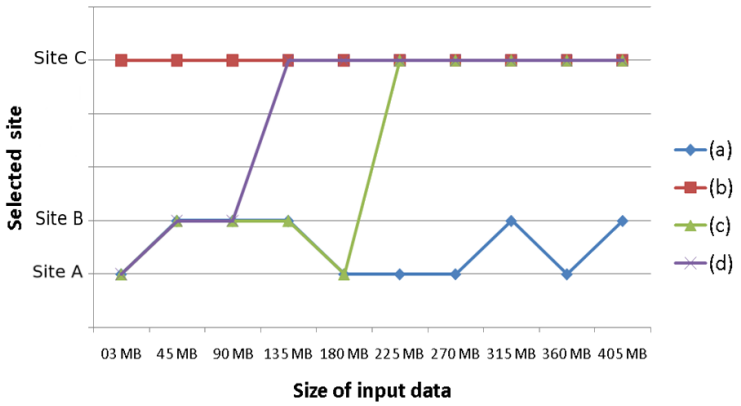


Fig. 6.5: Selected site as a function of input data size.

Finally, Table 6.2 summarizes the average values obtained for job execution time and the number of file transfers that were required. As expected, policy (a) obtained the worse results both in average job time and number of transfers. Policy (b) avoided all data movement but it was not optimum when input files were small. Policy (c) and (d) took wasted time into ac-

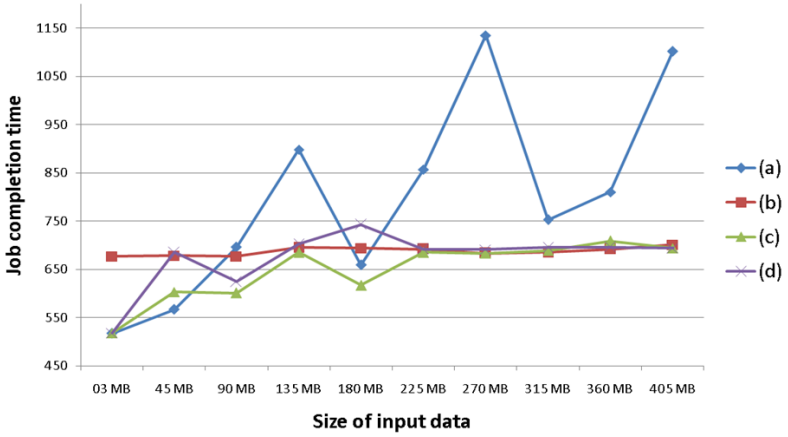


Fig. 6.6: Turnaround job time by policy.

Table 6.2: Average job time by policy.

Algorithm	Average job time	Number of transfers
(a)	799.4	10
(b)	687.4	0
(c)	648.4	5
(d)	674.0	3

count, with (*c*) obtaining the absolute minimum of average job time and (*d*) trading a slightly worse result for a reduction in the number of transfers.

6.4. Coordinated Workflow and Data Placement

6.4.1. Data Placement System

We have shown that GridWay scheduler can be enhanced to use data location information for its scheduling decisions. How this information is actually used is responsibility of the submitter (via the requirement/rank expressions). This solution is more flexible than that provided by previously existing EMI's WMS since the VO is the one deciding how to weigh data transfers versus processing power or queuing times. For a general purpose metascheduler, this is an important property. However, it is clear that the resulting system is not perfect. Its most obvious limitation is that it cannot estimate how long a transfer will take. If an infrastructure such as WLCG offered a service providing reliable link information, this might be added as a source for GridWay to compute foreseen transfer delays.

The other main problem with the described approach has been already outlined earlier. It is manifestly suboptimal that each individual job per-

forms the transfer of the files it needs. This task is better achieved by a data placement system. Moreover, a DPS is probably informed about other factors, such as free space at the SEs, VO policies for data replication and so on. Therefore, we think that the system could be greatly improved if, instead of querying an external service for link status information, GridWay could be enabled to query a DPS and also let it take charge of the necessary transfers.

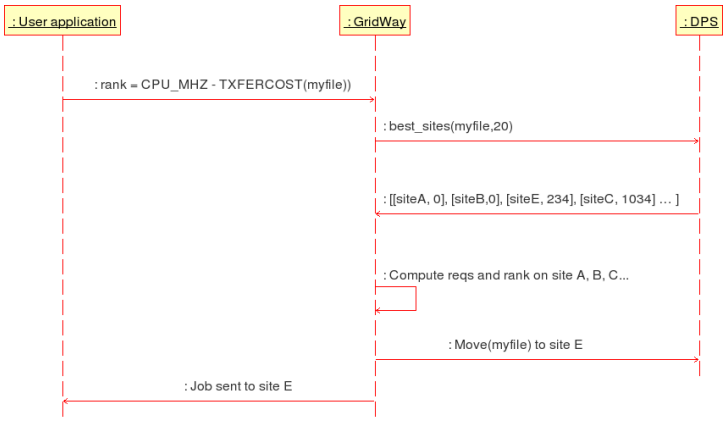


Fig. 6.7: Integration of a data placement system as information source for GridWay.

The integration of a DPS in the scheduling process is illustrated by Figure 6.7. We can see how GridWay would use the `best_sites` query to ask for the costs associated to a given file (`myfile`) and the reply would be the list of prioritized sites, each with an assigned cost for the transfer of the files. In the example, some sites (A and B) already hold the file and are given a null cost, while others (C, D, E...) could receive the file incurring in certain cost, calculated based on link qualities, free storage, policies, etc. At this point, GridWay could perform the match-making of job requirements and calculate the rank for each resource in the normal way, but taking these transfer costs into account. Optionally, GridWay could return the chosen destination (D) to the DPS, so that it performed the data movement on behalf of the user. Such a `best_sites` service would replace the current query to the DLI.

6.4.2. Workflow Management System

If the DPS service to query for transfer costs was available, the indicated approach for GridWay's scheduling would probably offer the best possible scheduling for a single job. This does not guarantee, however, that the adopted decision produces the best results from a VO point of view. Consider the following example: A user submits 500 jobs, each of which requires the same

set of 10 input files, at a single site. This first site has only a few free CPUs while another site with many idle nodes is ready to accept a replica of the files. For a single job, it is probably best to submit it to the first site because transferring all the files would cause too much overhead. Nevertheless, for 500 jobs, the time required to move ten files would be negligible compared to let the 500 jobs queue at the first site. It would be more intelligent to replicate the data while the first jobs are starting to run and schedule the rest to the second site.

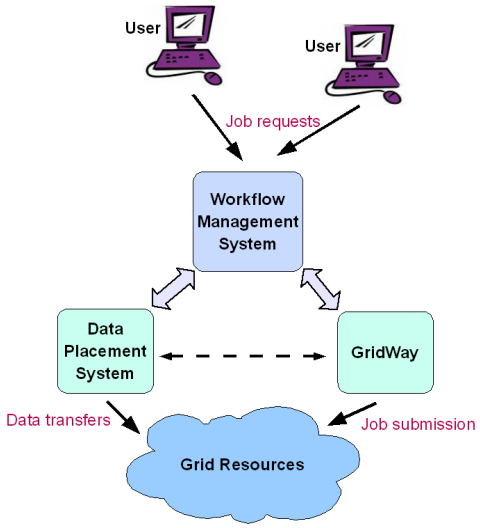


Fig. 6.8: Coordinated scheduling of jobs and data using a workflow management system.

A possible solution to this problem would be to use a higher-level component, perhaps a workflow management system, to take longer term decisions regarding data replication and planning of jobs destination. This component would get job requests as input, data information as feedback from the DPS and VO rules as policies. It would then schedule data replication tasks, which the DPS would execute and, by setting the requirement and rank expressions, instruct GridWay to allocate jobs as appropriate. The DPS and the scheduler would work independently. The organization of such a system is sketched in Figure 6.8.

Part of this scheme is actually already applied in WLCG. As discussed in Section 3.4.2, the workload management system of ATLAS VO is capable of requesting the replication of certain datasets if a large number of submitted jobs plan to run on that data. CMS is also taking steps in the same direction. This cannot be considered to be an optimization of the whole problem of job scheduling and data placement but a helpful optimization of the otherwise decoupled approach to both problems.

6.4.3. Decoupled Systems

The problem with the high-level workflow management system depicted above is that it is really complicated to make it work fine in practice. As already indicated, grid information is very dynamic and not always reliable, VO policies can get very complex and, all in all, data management is a very intricate science, with frequent rule exceptions. In WLCG's reality, the high-level strategical and operational decisions regarding data placement and job scheduling policies are taken by humans, not by an expert application. Even if this is very costly in terms of operational effort, it is sometimes the only way to make the whole system work appropriately.

Given the practical difficulties presented by the coordination of job scheduling and data movement, a more pragmatic approach may be followed. As discussed in Section 3.3, such a strategy is proposed by Ranganathan *et al.*, which suggests asynchronous replication of highly used datasets and simple submission of jobs to data [41]. WLCG VOs are currently following a similar path by applying popularity-based automatic replication and so reducing the human effort required by data distribution. In addition, thanks to the improvements in WANs and the reduction of I/O latencies, VOs can now complete this solution with restricted access to remote data. This is used to access missing files or circumvent overloaded sites by diverting jobs elsewhere and performing remote processing. The advantage of remote access over data replication is that it does not consume free storage space in the local SE.

WRAP-UP: *This chapter has discussed why location of input data must be taken into account when allocating data-intensive jobs, not only to optimise jobs efficiency, but also to avoid excessive data replication and the problems that this entails. This idea motivated the development of an enhanced GridWay metascheduler, which can incorporate data requirement information into the scheduling decisions in a flexible way (able to apply different VO policies).*

However, achieving an optimum data-aware scheduling is not a simple task. Information about job duration and transfer times is usually not reliable and data placement policies are complex. It seems clear that DPS's should take care of transfers but it is not so obvious whether a superior entity should coordinate these with job scheduling or, on the contrary, should data placement and workload management be completely independent. Current trends in WLCG lie between these two extremes. Files are automatically replicated and placed according to their past popularity but the scheduler is able to suggest the replication of some data to the DPS based on the requirements of queued tasks.

Chapter 7

Late-binding Overlay

Previous chapters have stressed that efficiently accessing data is one of the main challenges of data-intensive workflows in grid computation. Not only for data transfers between remote locations (through the WAN) but also considering the data access within a site. It has been observed that when many jobs try to access a Storage Element concurrently, this may cause responsiveness problems in the SE, possibly creating a bottleneck for the workflow processing and delaying its completion.

We have also discussed the advantages of late-binding overlay systems for the scheduling of computational tasks in the grid and the reasons why this solution has been chosen by all the large WLCG VOs to replace the traditional submission mechanisms. In addition, we have seen how the process of assigning real jobs to pilots can be used to implement a fine-grained micro-scheduling, in which we select exactly *where* a task is run.

Building on all these concepts, the *Task Queue architecture* was proposed as a new late-binding scheduling system, in which pilot jobs implement a data cache with the aim of optimizing the access to data, improving workflow turnaround times and reducing load on the site's SEs. In order to take advantage of these pilot caches, the TQ (*Task Queue*) must assign tasks to the nodes holding their required input data, so micro-scheduling is a requirement for the system.

This work was originated in the context of the WLCG, for the use of the CMS experiment. At the time, the CMS Tier-0 workflows were the main target of the new system, due to several reasons. Firstly, the latency of the Tier-0 operations is critical (derived calibration constants are required to reconstruct new data and disk buffers fill up if data is not processed in a timely manner), so reducing it was a primary objective. Secondly, the SE at the Tier-0 had been observed to occasionally become a bottleneck for the workflow runs, therefore the data cache promise was a highly desired feature[1]. Finally, the Tier-0 WNs were dedicated to CMS, which means

[1] As discussed in Section 3.2.1, the improvements in SE technology, the introduction of

that this was a controlled environment, where very long pilot lifetimes were guaranteed and large disk space could be allocated to each pilot, enabling very effective data caches.

Even if the initial target of the new pilot overlay was to run CMS workflows (especially, at the Tier-0), the TQ architecture was designed to be flexible enough to run other kind of jobs. Moreover, in order to evaluate the system under different configurations and environmental circumstances, we built a setup in which arbitrary workflows could be run and different storage access conditions could be simulated.

This chapter discusses the proposed TQ system, from its architecture, in Section 7.1, to the details of data caching, in Section 7.2, and micro-scheduling, in Section 7.3. An evaluation of the whole system, both in the CMS Tier-0 context and in more general scenarios is given on Section 7.4.

7.1. The Task Queue Architecture

7.1.1. Overview

The Task Queue system proposes to build a late-binding overlay for job submission and scheduling using pilot agents. The pilots are submitted to the grid resources using traditional mechanisms and retrieve real workload at runtime. The pilots implement a data cache on their local disk to avoid accessing the site storage services as much as possible. A fine-grained matching of pilots (and the resources they represent, including cached data) and the CMS tasks (real jobs) is performed by the central TQ server upon job request, enabling the micro-scheduling of the workload. Figure 7.1 shows a schematic representation of the proposed architecture.

Since the original target application for the TQ system was the CMS workloads, the figure shows the classical components of CMS job submission, *ProdAgent* (older) and *WMAgent* (newer), as clients of the TQ. Instead of having these components submit jobs directly to the grid sites, as they normally would, new TQ plugins are added so that jobs are instead enqueued as tasks into the TQ component. The new architecture did not aim to replace the existing systems, but offered a way to enhance those with a new solution. From the point of view of Prod/WMAgent, the TQ is just another resource to submit jobs to. Conversely, the TQ manages tasks independently of the client that enqueued them. Notice also that the TQ is not a workflow management system. It is the client of the TQ who creates tasks

the EOS system at the Tier-0 and the optimizations of the I/O access patterns of CMS applications have made most storage access problems at Tier-0 go away. For this reason, the pilot data cache is no longer a priority for CMS, though the general conclusions of our study are still valid. Moreover, we consider that our architecture could be helpful at many different scenarios, as discussed throughout the document.

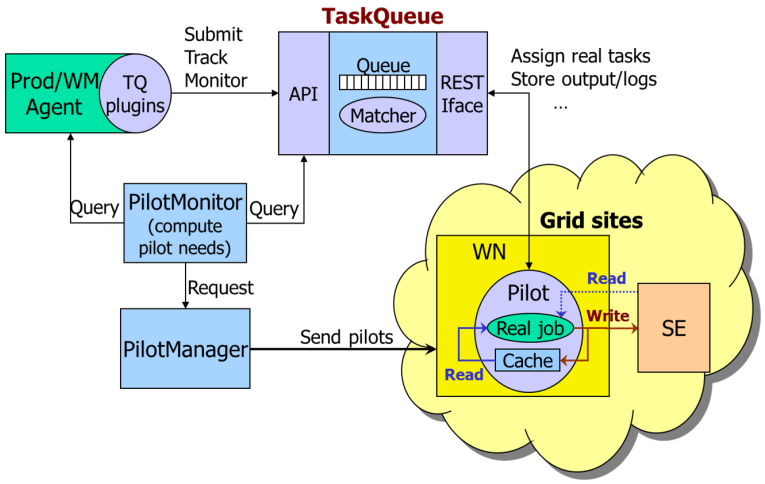


Fig. 7.1: Overview of the Task Queue architecture.

and sets their dependencies. The TQ gets these and attempts to make the best possible use of the available resources. The TQ is a generic solution, not tied to the CMS machinery.

The TQ component is the core of the late-binding overlay. It keeps track of all real jobs waiting to be scheduled and of their requirements. It also knows about pilots running on the grid resources and about the data they hold in their caches (file catalog). Finally, the TQ is the place where the logic for the matching of tasks to pilots is performed. It is no surprise, then, that this component gives the name to the whole system. In the following, we will sometimes use the term *Task Queue* to refer to the complete architecture and sometimes to designate this particular component (the context should make the implied meaning clear).

The *Pilot Monitor* component periodically checks the status of pilots and tasks in the system and computes the number of pilots to submit to each site. Then, it requests the *Pilot Manager* to perform the submission. The Pilot Manager is responsible for actually submitting the pilots, using the traditional interfaces (i.e., direct job submission).

Once running on a WN, the pilot job contacts the TQ, using a REST (*REpresentational State Transfer*) interface. It firstly registers itself, providing hostname and site information. Then, and for the duration of its job slot, the pilot retrieves tasks for execution. These tasks often need to access data located at the site's SE or, if possible, directly from its cache on disk.

The following sections provide more details about concrete aspects of the TQ architecture. In addition, more in-depth information on the implementation and the internal architecture of the different components is provided in Appendix A.

7.1.2. Pilot Management

The number of pilot jobs that should be submitted to a site depends on the amount of jobs that are waiting in the TQ for that particular destination and the number of slots at the site. In addition, certain predefined thresholds are defined according to the resources at each site and policy constraints. The Pilot Monitor component, responsible for monitoring the state of the submitted pilots, calculates the required number of additional pilot jobs—within the thresholds—and requests the Pilot Manager to submit them.

In order to perform this calculation, the Pilot Monitor counts with the information of the state of submitted pilots (running or queued) and the feedback provided by the TQ regarding the number of queued tasks, their site requirements and the number of idle[2] and active pilots.

Detailed information on the pilot release algorithm used by the Pilot Monitor, as well as on the per-site thresholds applied to the calculations, is given in Appendix A.3.

7.1.3. Pilot Job Operation

Once a pilot job has been scheduled on a worker node within an execution cluster, it will perform some initial checks to verify that the environment is ready to run real jobs. In the CMS case, among other things, it verifies that the CMS software and the local file catalog are accessible. If all the tests are passed satisfactorily, the pilot contacts the TQ at a well known endpoint and registers with it. As a result, the TQ provides the pilot with a unique ID. This ID will be used to identify the pilot in subsequent requests.

The pilots use the TQ's REST interface to communicate with it. All communications are started by the pilot, so that no incoming connections into the worker nodes is needed (this is often a security requirement of grid sites). Every request and response is conceptually considered a *message*, encoded in JSON (*JavaScript Object Notation*) [108] and follows a well defined protocol. Every message is marked with a certain *type* and includes a series of *fields*, which comprise a label and some contents. The type of a message determines which fields are compulsory for the message (additional arbitrary fields are usually accepted, though probably ignored) and what type of response is acceptable for a given request.

Once the environment has been setup and the registration completed,

[2]Idle pilots are those registered pilots that are not running a real job.

the pilot job is ready to request its first task. The pilot informs the TQ about its characteristics and the TQ selects the more adequate task for it. If there is no task available, the pilot just goes to sleep for some time before trying again. This loop continues until a task is found or some configurable threshold is reached and the pilot decides to shut down. When a task is found, the pilot is informed about its specifications and is provided with a task-specific URL (*Uniform Resource Locator*) to download its associated sandbox (containing the real job code and any necessary additional files). The pilot also receives a second URL where to upload the task's completion report and any produced log files. At this point, the pilot runs the real job in a separate process and waits for its termination. When done, the pilot informs the TQ about it and asks for another job.

The pilots continue executing tasks until they consume all the time allocated by the site's batch system. At that point, they inform the TQ and exit. During all this time, the pilots send periodic *heartbeat* messages to the TQ, so that this knows that they are still alive.

7.2. Data Caching

As discussed previously, the pilots composing the Task Queue overlay use the available disk space at the WNs to build a data cache in order to avoid (or alleviate) the problems that excessive concurrency may cause in the storage elements of the grid. Files staged in or produced by tasks are stored in the cache and can be reused by subsequent tasks. The pilot manages the cache space, respecting a pre-configured threshold for the maximum size it can reach. If there is not enough free space for the addition of new files or if a periodic check detects that the cache has grown too big, old files are removed until the used space falls below the maximum threshold. For every file added or removed from the cache, the TQ is informed, so that the central catalog is updated.

Regarding the cache replacement, our pilot jobs currently use a simple LRU (*Least Recently Used*) policy, although this is configurable. Other approaches might be more useful, depending, e.g., on the application at hand, but we have not studied this topic.

Real jobs run by the pilots will by default look for their input files in the site's SE. In principle, they would need to be modified so that they check if the required files are already available in the local cache. In the case of CMS workloads, however, this was not necessary. CMS tasks use a mechanism called TFC (*Trivial File Catalog*) to discover how (which technologies) and where (at what location) to access the data. The TQ pilots modify the task's copy of the TFC, so that, for cached files, the modified TFC instructs them to read from the local disk, instead of resorting to the SE.

7.2.1. Per-host Cache Sharing

Given the increasing density of CPUs in modern computers, it is everyday more common that grid's WNs run a relatively high number of concurrent jobs. It is therefore quite possible that several TQ pilots end up running on the same WN. For CMS, this is especially true in the Tier-0, where nodes are dedicated to the VO, so all slots in a worker node might be simultaneously running TQ pilots.

Since a cache is much more effective if data can be accessed by different pilots (increased cache hit ratio), we explored the possibility that all the pilots on a given host could share their cache spaces. This was achieved by having pilots creating Unix hard links to the cache directories of other pilots. In this way, all the files are, in practice, part of all the different caches, and they remain in the system even if their original owners delete them. A file is only removed for real when the last link is deleted.

In order to create these links, pilots are informed by the TQ about other pilots at their host upon registration. In addition, if the list changes, the TQ provides updated information to the pilots in response to their next heartbeat message. We called this configuration *per-host cache*. In this case, the total cache space available to each pilot in a WN is a dynamic value that depends on the number of pilots running at the same host. The formula to compute this value is given by Equation 7.1:

$$Cache\_Size = Max\_Space \cdot Num\_Pilots - Min\_Threshold \quad (7.1)$$

Though this configuration was implemented and tested (as shown by the results of Section 7.4), it presents some practical problems, such as the impossibility to create hard links between different file systems or, e.g., in non-Unix machines. In addition, it complicates the logic of operation. Since, as we will see in Chapter 10, we later developed a DHT-based system to share files among all the pilots, even if running on different hosts, we abandoned the per-host cache implementation.

7.3. Job Matching. Micro-scheduling

As discussed in Section 5.3, micro-scheduling is a requirement for an effective usage of the data cache built on the TQ's pilots. We also indicated that micro-scheduling can potentially be applied to situations that go beyond the scheduling of jobs to their required data (e.g., balancing I/O- and CPU-intensive jobs or matching pilots according to the number of cores in a WN). However, the matching of the tasks to the pilot nodes holding the required data was the primary motivator for our work in this field and it is still an excellent example of its application. Moreover, the efficacy of our data

cache (the cache hit ratio) will serve as a meter for the quality of our micro-scheduling algorithm. We will see this in our evaluation of the system, in Section 7.4.

7.3.1. Micro-scheduling in the TQ Architecture

Tasks are enqueued in the TQ together with some requirements, such as its preferred site for execution (usually, because they require some data at the site's SE) or their list of input files (LFNs). The requirements are stated as a python logical expression that the TQ evaluates to true or false. Only pilots satisfying the requirements are considered for running. In addition, an arithmetic ranking expression is also associated with the tasks to evaluate possible pilot-task pairs with a numeric value. The ranking expression can make use of data evaluated at runtime, such as the list of files in the cache's host, and can even formulate things like *if the hostname of the worker node was included in the requirements, then add 1,000 to the rank value.*

When a pilot job starts execution, it contacts the TQ and asks for a task. The request to the TQ includes attributes such as the pilot ID, the execution host, the site's SE, the remaining TTL (*Time-to-live*) and the contents of the data cache. With all this information, the TQ evaluates, for all the tasks in the queue, which ones are eligible to run in the requesting pilot. Then, for these, it computes their rank value for the requesting pilot. The task with the highest associated rank is assigned to the pilot.

Notice that since the TQ serves one pilot request at a time, the assignment selects the best task a pilot can match, but it cannot take into account if other pilots would be better suited for that task or what is the globally optimum assignment of tasks and pilots. This is clearly a suboptimal solution. Our more recent TQ architecture uses a different approach, as discussed in Section 10.3.1.

Regarding the task data dependencies, there are two possible approaches for the matching procedure. We could set the input files as requirements of the task, so that it would not run until a pilot holding the files was found or they could be included in the rank, so that it would preferably run on a pilot with the files (or with the highest number of files) but, otherwise, would just run in any other pilot. The latter option seems superior, since the pilot with the files might take very long to become free or, worse, it might not exist. In any case, in the typical workflow in a scientific grid, if a real job does not find its files in the cache, it can always find them in the SE.

Notice however that by letting tasks run as soon as they are requested by a pilot, even if this does not hold their required input files, the cache hit ratio is severely affected. In order to fight this to some extent, our matching algorithm implemented the following `waitForData` policy: prior to scheduling a task to a pilot without the required data, the TQ checks if other *idle*

pilots do hold the data; if that is the case, the task is reserved. This resembles the *delay scheduling* discussed in Section 3.4. The contexts are however different since our pilot cache does not provide multiple replicas of each file and we cannot assume that tasks are short. For these reasons, we only wait if a pilot with the required file is currently idle.

The *waitForData* policy is very demanding for the TQ, since it has to compare the cache of all the pilots to the requirements of every task, *for each pilot request*. This is not scalable and was discarded in favour of a distributed cache, as we will see in Chapter 10, but it was used for the evaluation tests of the centralized TQ architecture presented in Section 7.4.

7.4. Evaluation

7.4.1. Tier-0 Tests

Since the TQ architecture was motivated by the CMS use case, the first objective of our evaluation was to validate its suitability to run CMS workflows from start to finish, as well as to analyze its performance in comparison to the previously used architecture. This assessment was performed by running a set of tests that executed real CMS's Tier-0 workflows at a dedicated computing queue at CERN.

The Tier-0 workflow consists of three main steps: *Repacker*, *PromptReco* and *AlcaReco*. The Repacker jobs reformat the binary data from CMS's data acquisition system and split the output into different primary datasets based on physics information. The PromptReco jobs take this output as their input and perform an initial reconstruction into usable sets of physics data (e.g., particle properties or trajectories). The AlcaReco jobs filter the data produced by the PromptReco jobs and perform some processing on the resulting subset. Their output is used to dynamically align and calibrate the CMS detector.

In each step, several jobs are created, depending on the number of physics events in the input files. Each job produces a relatively small output dataset compared to its input. It is inefficient to store and transfer small files to a tape-based central SE, therefore, each step has a special *merge job* that gets the output from multiple jobs and merges them into a single file, which is then transferred to the SE. The creation and execution of all the workflow steps is driven by data. The workflow starts execution whenever a new file requiring processing is available. The subsequent jobs are created according to the system policies, workflow rules and data availability.

7.4.1.1. Setup

A series of experiments have been conducted at CERN's Tier-0 infrastructure[3]. For these experiments, a testbed comprising a cluster of ten machines has been used. Each of these machines is capable of running four jobs in parallel. The CMS Tier-0 reconstruction workflow is used as a sample workflow in these experiments. This workflow generates a total of 172 jobs, requires 83.41 GB of input data and produces 112 GB of output data. The jobs access CERN's storage infrastructure (Castor) to store and retrieve data.

All real jobs are enqueued in the global scheduler of PA (*ProdAgent*) but two different configurations have been used. In the first one, the traditional setup, PA submits the jobs directly to the local batch system. In the second one, the newly developed pilots overlay is set as resource destination for the real jobs (tasks) and pilots are automatically submitted to the batch system.

7.4.1.2. Results

The results of the described experiments are presented in Figure 7.2. We can see that the workflow turnaround time has been reduced by 4 % when using the proposed system. This decrease is mainly due to the reduction of the delays incurred in job submission and job status notification since the pilot-based approach reduces the latencies (this will be further discussed in Section 7.4.2.3). The improvement is small because, given the size of the testbed, most of the jobs had to be queued in the batch system and, thus, the impact of the shortage of the job submission delays was not noticeable.

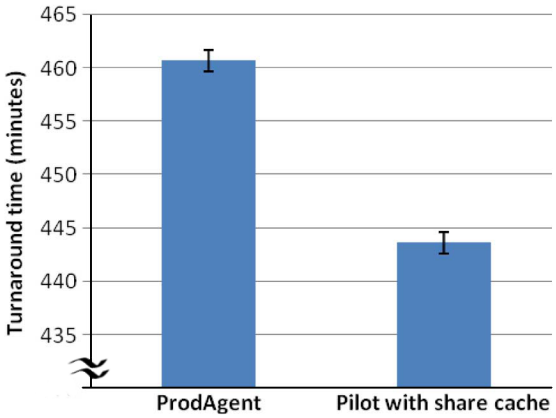


Fig. 7.2: Turnaround time for traditional submission and new Task Queue.

[3]The experiments in Section 7.4 have been repeated several times and, unless otherwise stated, all the values shown in the presented figures are the average of the measurements while the error bars show the standard deviation of the mean.

In these tests, it was not possible to measure the behavior of the proposed system against parameters such as job failure rates, queuing times or data access latencies. This is so because there was no additional load on the SE instance used for these tests and it was not practically feasible to artificially alter its access conditions. For this and other reasons, additional tests were run in a more controlled environment, at the CIEMAT (*Centro de Investigaciones Energéticas Medioambientales y Tecnológicas*) institute. These tests are discussed in Section 7.4.2.

7.4.2. CIEMAT Tests

With the purpose of demonstrating the general applicability of the TQ system and to gain flexibility, a new workflow engine, capable of running arbitrary workflows, was implemented and a testbed was arranged, at the CIEMAT institute, in Madrid, Spain. This allowed us to run a series of simulation tests that evaluated the impact of different storage access conditions, data dependency patterns and caching configurations in the performance of the new architecture.

7.4.2.1. Testbed

For the experiments, a new, lightweight workflow engine was developed to replace the role of the PA. This system produces workflows of configurable characteristics. It creates a series of jobs, enqueues them as tasks in the TQ and monitors their evolution to produce new, dependent jobs, as required. Figure 7.3 shows a simplified diagram of the complete testbed. The image is very similar to Figure 7.1, with the PA having been replaced by the new non-CMS workflow management system. In addition, the real jobs are not typical CMS tasks, but a series of scripts specifically developed for the tests. The details of this testbed are given in Appendix A.6.

The non-CMS jobs do not really consume CPU cycles or stress the local SE. On the contrary, they pretend to spend time processing or accessing storage, but in reality they just wait, idle, on the WN. By running these specially crafted jobs, we are able to easily modify their configuration in order to simulate different external conditions. In addition, they are appropriately instrumented to provide all the monitoring information in their completion reports, so that the external machinery can automatically produce the desired statistics and generate plots.

7.4.2.2. Configurations

Workflows. In these experiments, 120 concurrent pilot jobs were run. The data-driven nature of CMS workflows was achieved by using *steps*. A workflow is divided in such a way that jobs in one step depend on the output

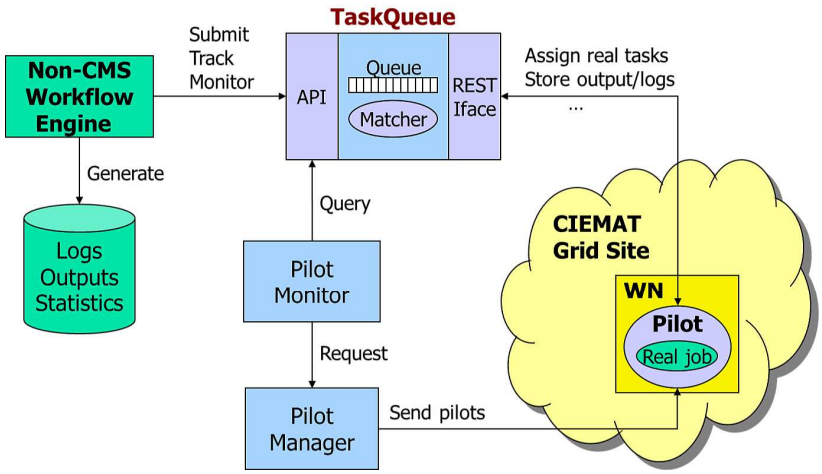


Fig. 7.3: Architecture of the non-CMS Task Queue testbed, at CIEMAT.

produced in the previous one. For the tests in this section, only two steps have been set up (*step*0 and *step*1). The jobs in *step*0 are generated and scheduled first. They produce output files, which are stored in the SE and in the local cache, if available. The jobs in *step*1 are only created when the files they depend on have been produced. They will consume those files from the SE or, preferably, the cache.

Three types of workflows have been used in these tests. They represent three different types of data dependencies generally found in data intensive scientific workflows and, particularly, in the CMS Tier-0 workflow. Each job in these workflows produces a file of size 700 MB. The workflow types used in our tests are the following:

- *Serial chain* (labelled as $W1$): corresponds to a one-to-one dependency. In this case, 80 jobs are run in *step*0, producing one file of output each. This is followed by another 80 jobs, in *step*1, each depending on one output file of *step*0.

- *Splitting* ($W2$): each job in one step is followed by several jobs in the following one (one-to-many dependency). In our tests, there are 40 jobs in *step*0 (with one file of output each), which are followed by 80 jobs in *step*1. Thus, two jobs consume parts of the same input file from *step*0.

- *Merging* ($W3$): This corresponds to a many-to-one dependency. In this case, *step*0 is composed of 80 jobs, producing 80 files of output (one

Table 7.1: Combination of delays and failure factors.

Config.	Delay	Failures	SE Load/Condition
$d1f1$	$d1 = 0.01$	$f1 = 0$	Low/Normal
$d2f2$	$d2 = 0.15$	$f2 = 0.03$	Moderate/Medium
$d3f1$	$d3 = 0.50$	$f1 = 0$	High delay but no failures
$d3f3$	$d3 = 0.50$	$f3 = 0.1$	High/Worse

each), which are consumed by 40 jobs in *step*1. I.e., each job consumes two files, from two different *step*0 jobs.

Storage Access Conditions. In order to analyze the behaviour of the traditional and the new architecture as a function of the SE responsiveness, we have equipped our non-CMS tasks with the functionality to simulate different storage access conditions. Two parameters may be configured for our tests: the delay factor and the failure rate. The first one indicates the additional delay that SE access operations will cause on the job (comparing to local disk access) while the second one represents the ratio of operations that are not completed successfully, causing the failure of the job and, thus, its resubmission. Combining these, we have considered four possible configurations. These are given in Table 7.1. In the following, their labels—$d1f1$, $d2f2$, $d3f1$, $d3f3$—will be used to indicate the conditions under which a test is run.

Note that the delay factors in Table 7.1 are average values. Each individual job may experience slightly different delays due to multiple uncontrollable causes, such as the exact disk server being accessed or the effect of concurrent operations of other users. We have modeled this by randomly selecting job delays from a Gaussian distribution with the selected delay factor for the workflow as its mean.

Caching and submission configurations. Three different job submission mechanisms have been used in the evaluation tests:

- *Direct submission*: Traditional submission (without TQ).
- *Pre-allocated pilots*: Pilot jobs are manually submitted and already running when tasks are injected into TQ.
- *On-demand*: Pilots are submitted by the Pilot Manager as a reaction to task injection (therefore, there will be an initial delay before injected tasks start to run).

For the cases where the TQ architecture is used, there is an additional setting: the type of caching used by the pilots. The different possible config-

urations are to use an independent *per-pilot* data cache, to build and share a *per-host* cache or, lastly, not to use any caching at all. In addition, if caching is applied, the TQ may be configured with or without the *waitForData* policy. In the following, all the tests where cache is used apply the per-host configuration, with the *waitForData* policy enabled (except for those in Section 7.4.2.5).

7.4.2.3. Submission Latencies

The plot in Figure 7.4 shows the number of running jobs as a function of time, for a $W3$ workflow and for the different submission mechanisms. There is an initial job submission delay for the direct and on-demand configurations. This delay is due to the scheduling latencies introduced by the middleware and the waiting time in the local batch system queues. However, this delay is not present when the pilots are pre-allocated and already waiting for tasks to run.

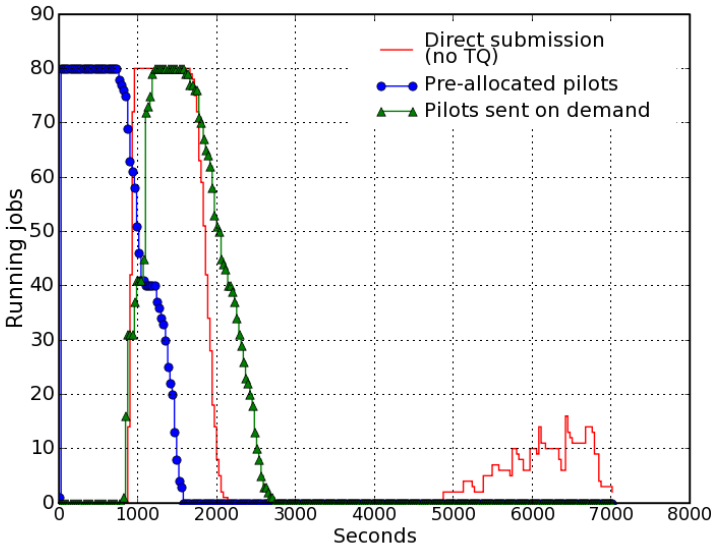


Fig. 7.4: Running jobs vs. time for different submission systems ($W3$ workflow).

Obviously, the results for pre-allocated pilots are the best ones. However, the on-demand configuration also shows a huge improvement over traditional submission. The reason is that *step*1 tasks can be run as soon as tasks from *step*0 are completed. When direct submission is used, though, there is a big gap between steps. This delay is introduced by the middleware: it takes a long time for the submitter to realize the first jobs have finished. And once this happens, dependent jobs need to go through middleware and batch

system queues again to start running. In this particular case, the reduction in the delays is more than 60 % for the on-demand pilot submission and more than 75 % when pre-allocated pilots are used.

Certainly, modern grid middleware behaves now better than when these tests were run. Also, we must note that these results are more spectacular because the real jobs used in these tests are very short (their length is comparable to the scheduling delays). However, the effect is real and it is one of the reasons why pilot-based systems are now predominant in large grid VOs. In any case, the main contribution of our TQ architecture with respect to other pilot systems is the caching and micro-scheduling capabilities. The following sections compare the behaviour of pilot systems with and without data cache.

7.4.2.4. Execution Times

Figure 7.5 shows the effect of stage-in delays on job execution times for different SE conditions, with and without cache, for a serial-chain workflow. In these tests, the stage-in time represents either the time for downloading a file from an SE or, if no download occurs, the time a job spends accessing the SE and reading the file while processing it.

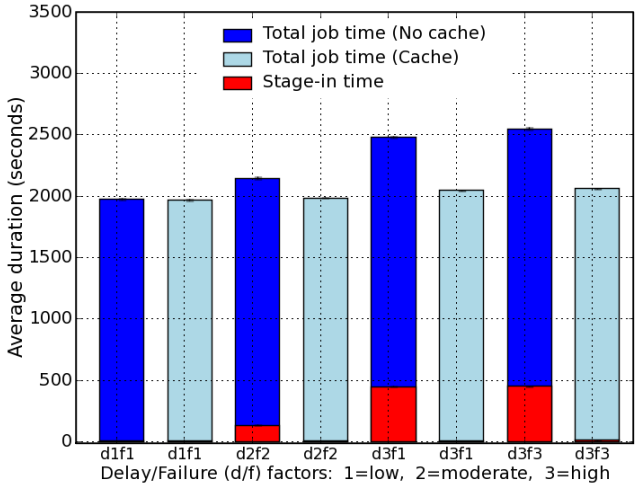


Fig. 7.5: Average job execution and stage-in time under different SE conditions, with and without data caching ($W1$ workflow).

The plot shows that when the pilot cache is used, the results are better, especially when the SE conditions are worse ($d3f3$). If the SE response is good, there is little gain in getting input files from the local caches. We can say that, to some extent, the use of the cache protects the jobs from problems at the SE. Notice also that by using the cache, the number of I/O

requests to the SE is decreased; therefore, its use may, in the first place, reduce the deterioration of the SE conditions.

Figure 7.6 shows the turnaround time of a complete workflow for the same runs than Figure 7.5. It is clear that, except for $d1f1$, the cache approach provides better workflow turnaround time than the no-cache approach. An interesting fact to note here is that an increase in the failure rate has a more significant effect on the turnaround time than an increase in the delay factor. This is due to the fact that a failure in SE access causes a job failure, which triggers its resubmission and, therefore, provokes new scheduling and execution delays. This impacts the whole workflow makespan but has no effect in the average duration of individual jobs.

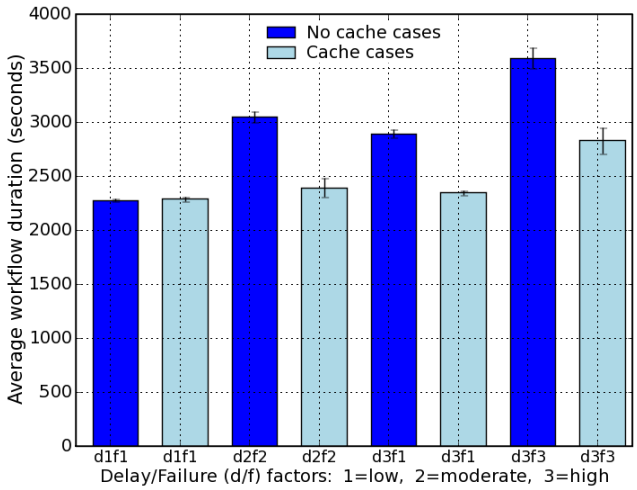


Fig. 7.6: Workflow turnaround time, under different SE conditions, with and without data caching ($W1$ workflow).

When the data cache is used, mosts jobs read their required files from the local disk. This causes a reduction of the data access latencies and of the number of failures during the stage-in time. However, the failures at stage-out (writing output files data in the SE) affect all jobs, with or without cache. This is the reason why the results obtained with high failure rates are significantly worse than those with rare failures, even for the pilots using the data cache.

Figure 7.7 shows the average job duration for different workflow types, under the worse SE conditions ($d3f3$). Looking at the stage-in time in each configuration, we see that the cache mechanism performs much better for a workflow where the jobs show one-to-one dependency ($W1$) because the jobs from $step1$ can be efficiently scheduled to the pilot jobs that hold the results of the jobs from $step0$. For the workflows $W2$ and $W3$, the cache hit ratio

is lower. In the case of $W2$, two jobs require the same data, but they will probably end up running on different pilots. In $W3$, jobs from $step1$ may be forced to read from two different pilots (reducing the chances that the caches of both are local to its host). On average, for all the three workflows, the system with the pilot cache behaves better than the one without it.

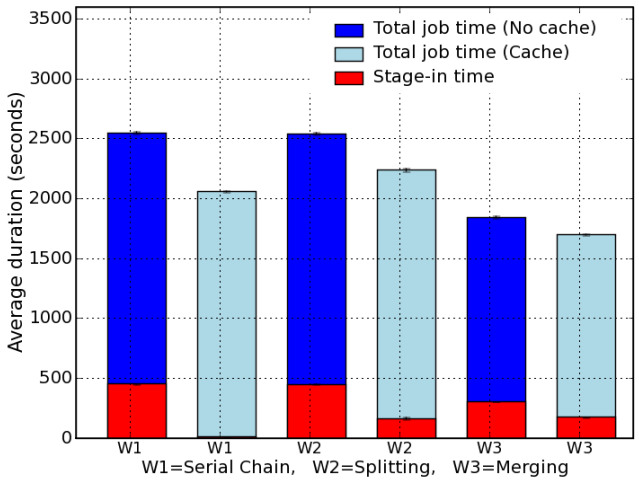


Fig. 7.7: Average job execution and stage-in time for different workflow types, under bad SE conditions ($d3f3$), with and without data caching.

Figure 7.8 depicts the cache impact on the turnaround times of different workflows, also under the worse SE conditions. Even if there is a greater dispersion of measurements, it seems safe to say that the pilots with data cache help to improve the workflow execution time when the storage resources are operating under stress.

7.4.2.5. Cache Hit Ratio

The cache hit ratio is the percentage of the read files that were found in the cache. It is an essential metric to evaluate the effectiveness of a data cache. Figure 7.9 depicts the cache hit ratio for different caching configurations and types of workflows.

For the serial chain workflow ($W1$), the per-host ($C1$) cache and the per-pilot ($C2$) cache with the $waitForData$ policy yield hit rates above 99 % because jobs in $step$ can be scheduled to those pilot jobs that are holding the file they require as input. When the $waitForData$ policy is not in use, the cache hit rate is severely reduced (23 %) because the TQ does not wait for the pilot jobs with the data to request the job. Consequently, a task is scheduled to a pilot job that may not be holding the required files in its cache. This effect is observed for all three workflows. In the absence of a

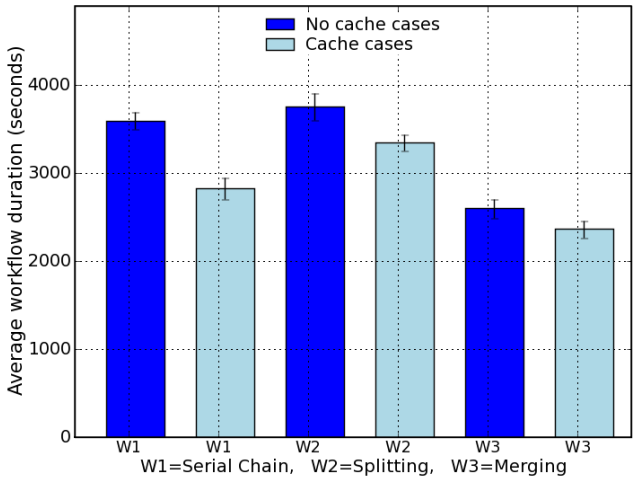


Fig. 7.8: Workflow turnaround time for different workflow types, under high SE load conditions ($d3f3$), with and without data caching.

global data cache, the $waitForData$ approach seems fundamental to achieve a good cache hit ratio.

The figure also shows that the efficacy of the per-host cache configuration reaches a 74 % hit ratio for the splitting workflow ($W2$), whereas the per-pilot configuration achieves only a 50 % hit ratio. Remember that, in $W2$, each $step0$ task produces two output files and two $step1$ tasks consume them. Since these two tasks are scheduled at barely the same time, they will, in general, be retrieved by different pilots, so one of them will not find its input file in the local cache.

In the case of the merge workflow ($W3$), each job in $step1$ requires two input files produced by two different $step0$ jobs, run on two different pilots. When the per-pilot cache is used, the maximum achievable hit ratio is obviously 50 %. The ratio could be higher for the per-host configuration. However, since $step0$ files are randomly scattered among the pilots, the probability that two input files are found on the same WN depends on the density of pilots per host and it is, in general, presumably low. In effect, in this particular case, we see that the per-host cache hit ratio is almost equal to that of the per-pilot cache.

WRAP-UP: *We have presented the Task Queue system, a pilot-based late-binding architecture, enhanced with data caching and micro-scheduling. By*

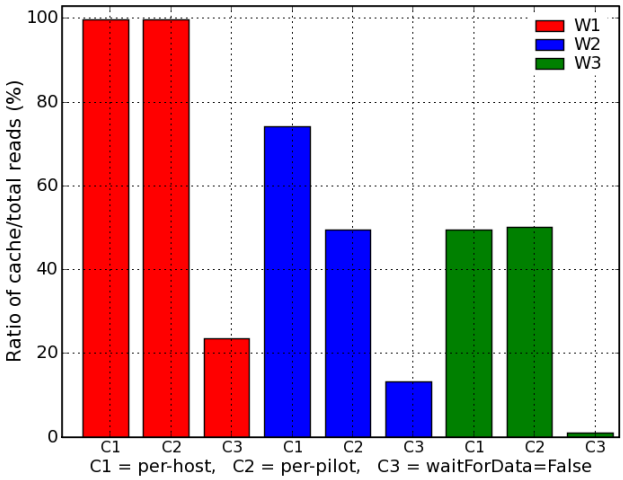


Fig. 7.9: Hit ratio for different cache configurations and workflow types.

evaluating the characteristics of each pilot against the requirements of the tasks and the configured rank expression, the TQ is able to assign the most appropriate task to each pilot. As a significant example, a pilot will preferably get a task requiring files stored in its cache.

A series of tests have shown that the implementation works as expected and that workflow turnaround times are decreased when, thanks to the cache, fewer accesses to the storage systems are required. This effect is more noticeable when the SEs are congested and so access latencies are higher.

However, the effectiveness of the pilots data cache is very dependent on the pattern of dependencies between jobs. For certain workflows, the results are not really satisfactory. Moreover, if the costly waitForData policy is not used, results are poor in almost all cases. These considerations and some scalability issues not yet discussed, led us to implement a truly shared cache among all the pilots in a given site. This will be introduced in Chapter 10.

Part III

DHT-based Late-binding Scheduling and Data Sharing

Chapter 8

Evaluation of Data Caching and Centralized Scheduling

Chapter 7 presented a new architecture for a pilot-based overlay with late-binding scheduling. This system was called *Task Queue* and introduced pilot data caching and micro-scheduling capabilities. Even if the system was validated to provide the expected functionalities and satisfy the essential performance requirements, some shortcomings were detected.

The following sections review the indicated weaknesses. Section 8.1 discusses the need for the pilots to share cached files in order to improve the effectiveness of the cache. Section 8.2 analyzes the scalability issues of the current centralized task matching algorithm, which have led to the development of a new distributed scheduling method. Closely related to this problem, the dependency of the whole system on the availability of the central server, which may constitute a SPOF, are studied in Section 8.3. Finally, Section 8.4 examines whether the existing task scheduling procedure yields the best possible association of tasks and pilots from a global point of view.

8.1. Distributed Data Caching

Chapter 7 pointed out that the achievable effectiveness of the data cache (measured by the accomplished cache hit ratio) was conditioned by the dependency patterns of the executed workflows. In some cases, it was not satisfactory at all. The per-host cache was always more effective that the per-pilot configuration but it still showed poor results for some cases. Moreover, the application of the highly demanding (in terms of CPU load put on the TQ) *waitForData* policy was a necessary requirement for achieving an acceptable cache hit ratio in all considered cases.

In order to overcome this, further reducing the required accesses to the storage systems, we decided to provide pilots with the means to share the

files of their caches. For this purpose, the pilots create a per-site DHT network and register themselves (and the files they own) with it. In this way, they can easily locate the pilot holding the file they need and fetch it. This will be discussed in depth in Chapter 10.

8.2. Scheduling Overhead

Another possible limitation of the TQ system is its scalability. In the architecture discussed so far, all pilots contact a single central server. Moreover, for each pilot request, the TQ component needs to scan all queued tasks to find the most suitable one. This centralized approach is doomed to suffer when operated at very large scales. The evaluation tests shown in Section 7.4 were run on only 120 concurrent pilot jobs but experiments like CMS manage a much greater number of simultaneous pilots and tasks. Actually, as we will see later, when we increase the scale of these resources, the TQ starts to show problems to run its centralized matching algorithm within a reasonable time limit. In order to solve this, we have developed a new distributed task matching algorithm.

This is actually not only a TQ's problem but of most late-binding systems. VOs using the late-binding approach usually aim to have a single central server from which tasks are pulled. This makes it easy to apply VO priorities and adapt them dynamically. It also means that each scheduling decision counts with all the available information (running pilots, queued tasks). However, as the number of pilots and tasks increases, so does the rate of requests to be served and—potentially—the complexity of the decision for each request. In short, the central queue of pilot-based systems may become a bottleneck for the scheduling and execution of large-scale workloads [109, 110].

If we compare this to the traditional early-binding paradigm, we find that, in that case, scalability is achieved by multiplying the number of schedulers. The problem with this approach is that, in practice, it is not feasible for all of them to maintain a consistent view of the resources at all times. Moreover, their actions are uncoordinated, which leads in occasions to chaotic competition for resources (some sites may be overloaded while others still own free resources). Besides this, as already stated, enforcing VO priorities becomes very complex since each submitter acts independently.

Existing pilot systems have already met this problem and have found different ways to tackle it. Tasks may be grouped and evaluated in bulk or pilots may be matched based on their site only [110]. In this case, micro-scheduling is effectively being restricted or even given up. Another possible technique is to assign tasks to pilots beforehand so that each request is tied to a particular task, so no real matching is carried out [63]. This means that, in fact, early-binding—rather than late-binding—scheduling is performed.

However, our TQ architecture aims to offer real fine-grained late-binding micro-scheduling (if for no other reason, for the establishment of the pilots data cache). Furthermore, the TQ's task assignment process supports not only task matching but also task ranking. This means that the pilot will not get *any* task whose requirements it fulfills, but the *best match*, according to a predefined ranking expression. Thus, semantics like *run task preferably on a pilot holding certain file* (best match) *but, otherwise, on any other pilot* (any match) can be supported. However, evaluating the requirement and rank expression on all queued tasks for every pilot request is a CPU-intensive duty, which can lead to TQ congestion and a rise in the time spent for each request. While waiting for the assignment of a task, pilots are idle and wasting CPU time. This delay is a *scheduling overhead*, which should be reduced as much as possible. This fact can also be deduced from the analytical model introduced in Section 5.2, as will be shown in Section 8.2.1.

It is because of these reasons that a new distributed task matching procedure was developed. This will be described in Chapter 10. As we will see, this procedure requires that pilots are able to submit broadcast messages to the nodes sharing their DHT network (in principle, those within the same site). Since the selected technology for the file sharing was Kademlia and this DHT (like many others) does not support broadcasting, we initiated a study on the best possible way to add this functionality to Kademlia. This resulted in a in-depth study of this problem (including several contributions), which is captured in the discussion in Chapter 9.

8.2.1. Impact of Scheduling Delay: Optimal Task Length

Section 5.2.1 presented a model for finite workflow execution in the grid. Considering discrete tasks and non-null matching delays, we reached Equation 5.14, which we recall, as follows:

$$L = avg(T_i^s) + avg(T_i^e) + C + C\frac{T^m}{T^t} \tag{8.1}$$

Looking at the terms of the equation, $avg(T_i^s)$ is—like in the continuous model—the average delay in the start of tasks, $avg(T_i^e) \approx \frac{T^t}{2}$ (limited by 0 and T^t in every case), C is constant and $C\frac{T^m}{T^t}$ increases with T^m and decreases with T^t. If we now consider the workflow makespan time (L) as a function of the task length, T^t, we find that, in general, L decreases for shorter T^t due to the T_i^e term (by reducing the size of the task, the slot can be filled more effectively). However, for very low values of T^t, L increases dramatically due to the $\frac{T^m}{T^t}$ term. In effect, T^m creates a limit to how short our tasks can be (below this value, the execution time is dominated by the time spent matching, downloading the task files, etc.). For a bigger T^m, the lower limit for T^t increases. Actually, in this simplified model, we can even

calculate the limit as the minimum of the function. Deriving L with respect to T^t and making it equal to zero, we see:

$$\frac{dL}{dT^t} = 0 + \frac{1}{2} + 0 - C\frac{T^m}{(T^t)^2} = 0 \tag{8.2}$$

$$T^t = \sqrt{2CT^m} \tag{8.3}$$

Equation 8.3 can be used in Equation 5.14 to obtain the minimum L:

$$L = avg(T_i^s) + avg(T_i^e) + C + \sqrt{\frac{T^m C}{2}} \tag{8.4}$$

And if we use the following expression:

$$avg(T_i^e) = \frac{T^t}{2} = \frac{\sqrt{2CT^m}}{2} = \sqrt{\frac{T^m C}{2}} \tag{8.5}$$

then we reach our final expression for the minimum L:

$$L = avg(T_i^s) + C + \sqrt{2T^m C} \tag{8.6}$$

8.2.1.1. Simulation

In order to better understand the model, we have run some iterative simulations and measured L as a function of T^t. We have fixed the total workload to compute, the processor speed and number of machines available for computation. We have also set an average value for T_i^s (this value does not affect the shape of the resulting plot, it only moves the plot upwards or downwards). We will use:

- $\frac{W}{\varphi} = 100,000 \quad ; \quad K = 100$
- $avg(T_i^s) = 75 \quad ; \quad avg(T_i^e) = \frac{T^t}{2}$

We have iterated through a range of T^t values, taking random T_i^s and T_i^e values for each one of the K slots and calculated their average to compute the resulting time L—according to Equation 5.14. We have done this for T^m values of 0, 3, 10 and 20. The resulting plot in Figure 8.1 shows total workflow makespan time versus the task length, for different values of match-making delays (T^m). In the plot, the minimum task length for each configured T^m is marked with horizontal and vertical dashed lines.

We can see that for $T^m = 0$ (no matching delay), there is theoretically no limit to how short we should aim our tasks to be. The shorter, the better, although it clearly makes no sense to have tasks of zero length ($T^t = 0$), so the optimum would lie in an arbitrarily small—but not null—task. For

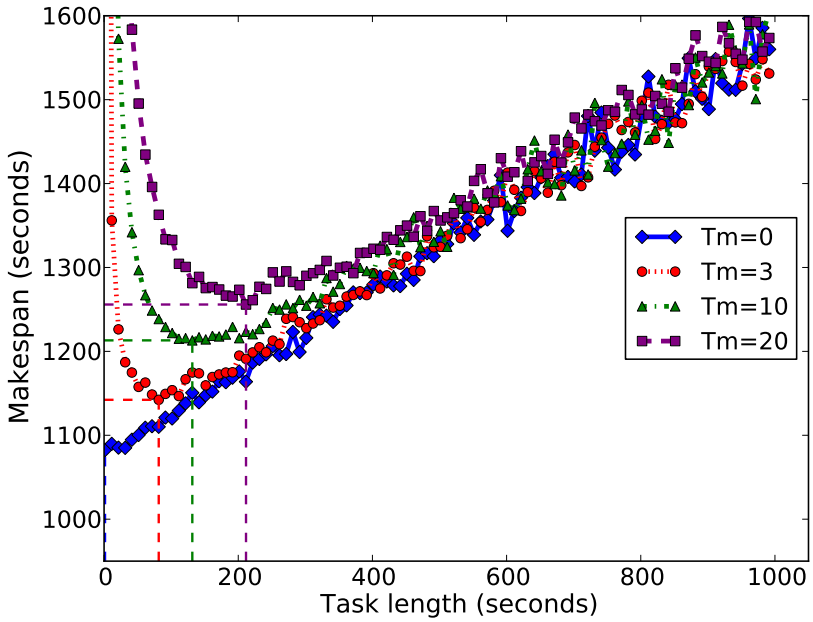


Fig. 8.1: Workload throughput model on early-binding approach.

the other cases, the simulation finds values that are close to the theoretical limit calculated by Equation 8.3. Table 8.1 shows both the theoretical and simulated values for these settings.

8.3. Pilots Autonomy

Due to the centralized nature of the original TQ architecture, the whole system is very dependent on the availability of the TQ component, causing it to be a SPOF. If this entity is not available, no pilot can register, acquire new tasks or inform about completed ones. This issue can be dealt with by setting high-availability configurations for the server or by making it

Table 8.1: Optimal task length and workflow turnaround time (seconds).

Matching delay (T^m)	**Model**		**Simulation**	
	T^t	**L**	T^t	**L**
0	0+	1,075	0+	1,075
3	77.46	1,152.5	81	1,147.4
10	141.42	1,216.4	161	1,210.3
20	200.0	1,275.0	201	1,262.6

redundant using database replication technologies.

The new architecture, which proposes a distributed task matching algorithm, aims to reduce the pilot's dependency on the TQ and thus increase the robustness of the system against TQ failures or connectivity problems. This will be discussed, including some assessments tests, in Chapter 10.

8.4. Micro-scheduling and Global Rank

In the centralized task matching procedure seen so far, the TQ receives one pilot request at a time and selects a task for it (maximizing the *rank* of the assignment). However, even if this work is done optimally (and timely), it is possible that sequentially choosing the best task for each individual pilot does not yield the best possible overall association. E.g., a pilot could retrieve the best task from its point of view, but that task may be even better suited for a different pilot. From a global perspective, we should try to maximize the *global rank*, i.e., the sum of all individual ranks. The problem of optimally allocating a set of tasks to a set of resources is actually known to be an NP-complete problem [111].

If we have a look at what other pilot systems do on this regard, we find that in general they also deal with pilot requests sequentially, so they should suffer from the same problem. The case may be different if they perform group matching or pre-assignments but, as already discussed, this is no really late-binding micro-scheduling, so, depending on how fine we require our assignments to be, the obtained global rank can be severely penalized. It is also worth mentioning the case of the glideinWMS system, which does not match one pilot at a time, but one task at a time instead [29]. Indeed, the glideinWMS *negotiator* receives collected information about tasks and resources and performs periodical matches but always iterating on tasks (and only considering idle pilots). It turns out that for our data cache use case, this method is actually better than iterating on pilots but that may not be the case in other situations.

In general, considering all (or several) pilots and tasks at each scheduling decision produces better results. That is what the new TQ architecture does.

WRAP-UP: *This chapter reviews the main limitations of the original Task Queue architecture (and of other late-binding systems): reduced cache hit ratio for some dependency patterns, scalability issues, excessive dependency on the central queue availability and sequential assignment of tasks to individual pilots. All these problems are tackled by the new architecture, discussed in Chapter 10, equipped with a distributed data cache and a cooperative scheduling procedure.*

Chapter 9

Broadcasting in Kademlia

In Chapter 4, we introduced DHTs, decentralized systems, frequently used in P2P systems, and Kademlia, a successful DHT, used in real applications like BitTorrent. Our interest in DHTs derives from our intention to implement a distributed data cache among the pilot nodes conforming the overlay of our Task Queue architecture. We chose Kademlia for the task, due to its convenient characteristics. Later on, motivated by the scalability issues found on the TQ centralized model, we decided to use the DHT routing functionality to implement a distributed task matching algorithm among the pilots. This took us to a investigation on the broadcast operation—required for the new task matching procedure—on Kademlia networks.

This chapter presents a detailed analysis of the possible approaches to the broadcast operation in the Kademlia DHT. Section 9.1 and 9.2 review the specific characteristics of Kademlia and the existing general-purpose proposals for DHT broadcasting, focusing on their applicability to Kademlia. Subsequently, Section 9.3 introduces our *bucket-based* broadcasting algorithm and Section 9.4 discusses the problems caused by churn and message loss and presents several techniques to fight them. Finally, Section 9.5 discusses the results of thorough experimental testing evaluating the shown algorithms and churn-fighting techniques.

9.1. Particularities of Kademlia

We described Kademlia in Section 9.1. As indicated there, one of the main characteristics of this DHT is that the distance between Kademlia entities is calculated as the result of the XOR of their IDs. Among other things, this means that unlike, e.g., in Chord, Kademlia distances are symmetrical and a node can initiate parallel look-up operations (using several contacts in the destination region). But this also implies that we cannot simply assume that two nodes whose IDs are numerically close are also close XOR-wise. For instance, the numerical distance between numbers 7 and 8 is 1, but their

XOR-distance is 15.

We have also indicated already that Kademlia routing tables are composed of a series of *buckets of K contacts*. Each bucket contains nodes within a certain distance range. In the simplest case, there is one bucket for each bit of node identifier. This can be seen as a binary tree: the ID space is successively split in two, increasing the number of bits in common with the node ID by one. The buckets can be made narrower in order to reduce look-up time at the cost of maintaining a larger routing table (considering b bits instead of 1 for each bucket). Each look-up iteration gets b bits closer to the target.

9.2. Existing Protocols

In Section 4.3, we described several general algorithms for DHT broadcasting. These protocols aimed to be applicable to any DHT (or, at least, to prefix-based DHTs). This section discusses their applicability to Kademlia.

9.2.1. Partition-based Broadcasting

As we saw, both k-ary and balanced partition-based algorithms set numerical limits on the region a node forwards messages to. This is based on the assumption that the first node in a region has a better knowledge of it that the original sender, because it is closer to all the region's members. This is true for DHTs like Chord, which uses numerical distance between IDs but, as we discussed above, it is in general not true for Kademlia, which uses XOR distances. In particular, nodes within a Kademlia bucket are closer to each other than nodes outside the bucket. However, nodes that lie on the edges of two adjacent buckets are very close numerically but quite far XOR-wise. This can be appreciated in Figure 9.1, which shows a binary tree representation of a network of 16 nodes and the initial broadcasting partitions for $\lambda = 3$ and $\lambda = 4$. We can see that, when $\lambda = 3$, node 1000 is included in the same region that node 0111 but not in the same one that node 1010, even if, considering XOR-based metrics, 1000 is much closer to 1010 than to 0111.

As a result, PB (*Partition-based*) algorithms will surely work fine with Kademlia if their key space regions map exactly to one or more buckets. Otherwise, the sender may not have a contact in the region even it partly overlaps with one of its buckets (because contacts may be located anywhere in a bucket). Our tests will confirm that with $\lambda = 3$ regions do not always map to buckets and there are cases where a tree branch is broken even in the absence of churn.

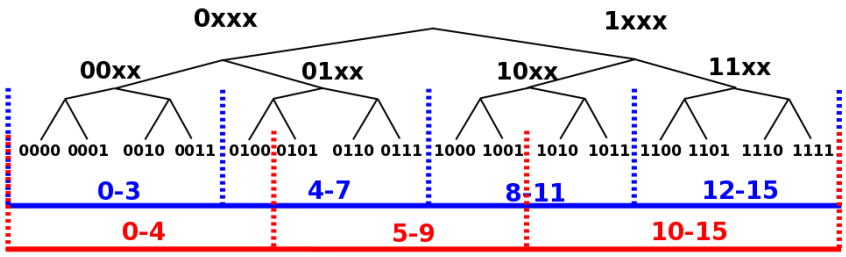


Fig. 9.1: Partition-based broadcasting trees for with $\lambda = 3$ and $\lambda = 4$.

9.2.2. Prefix-based Broadcasting

We already indicated that Kademlia can be described as a prefix-based protocol. Moreover, although described in different terms, Pastry's prefix-based routing is very similar to that of Kademlia (buckets can be mapped to entries in Pastry's routing table). There is however a difference between the two: when $b > 1$ bits are used, Pastry requires an additional routing phase to discover the closest nodes from those sharing the same prefix but with a different final digit. This does not happen in Kademlia, where a single XOR-based routing algorithm is used from start to finish.

The broadcasting algorithm described by Wahlisch *et al.* is meant for this kind of prefix-based DHTs and applied to Pastry [100]. The idea is to form a tree based on an increasing common prefix. The algorithm can be also used with Kademlia. The small difference in the last phase of Pastry's routing is not relevant when broadcasting, since all nodes in the last forwarding step will be messaged. In fact, if this algorithm is correctly applied in Kademlia, the obtained tree would be exactly the same as if we apply our BB (*Bucket-based*) algorithm, which we describe in the next section.

9.3. Bucket-based Broadcasting

The most natural way to divide the key space in Kademlia is bucket-wise. A broadcast initiator may simply send messages to a contact in each bucket. Recipients will forward the message to its contacts in the buckets within their own region. The identifiers of the nodes within the region that have already been contacted are included in the message so that broadcast is not sent to them again (to avoid loops).

Kademlia guarantees that, in stable conditions, there is at least one known contact for every bucket where real nodes exist. Moreover, closer buckets are narrower and a node knows all the peers in its containing bucket. Therefore, by letting nodes send a message to each bucket where they know a contact, a 100 % coverage is obtained.

The only tricky part in proving that this algorithm works fine comes

from the fact that buckets are different for each node (since they represent distances to the node). Thus, one must be sure that when a node sends a message to a contact in one of its buckets, this contact will be able to map this region to its own buckets to further divide it properly. We provide a proof that this is the case for Kademlia in the next section.

Authors of [102] try to make the broadcasting process clearer by applying a transformation by which every node forwarding a message becomes root (node 0) of its subtree. This is apparently the only difference with our proposal, but both approaches result in the same broadcast tree.

9.3.1. Demonstration for the Bucket-based Broadcasting

The BB algorithm only works if every region to which a node sends a broadcast message maps to a number of complete buckets of the recipient, being these buckets closer to it (XOR-wise) than to the sender. We will now prove that this is the case by showing that, given two nodes, A and B, with T being the width of A's bucket containing B, every node whose distance to A lies in that bucket has a distance to B lower than T. This means that B can just forward to nodes in its own buckets ranging from 0 to T.

In Kademlia, the distance between A and B is computed as $\overline{AB} = A \oplus B$. A node with ID P belongs to A's bucket $[X_1, X_2)$ if $\overline{AP} \geq X_1$and $\overline{AP} < X_2$. By construction, buckets have ranges $[j \cdot T, (j+1) \cdot T)$ where $T = 2^{(160-b(i+1))}$ for each j in $(0, 2^b)$ and i in $[0, \frac{160}{b})$. This means that their width is $T = 2^n$, for some n below 160.

We will use the following XOR properties:

(1) Let *a, b, n, k* be integers. If $a < k \cdot 2^n$, $b < 2^n$, then $a \oplus b < k \cdot 2^n$.

(2) Let *a, b, n, k* be integers. If $a \geq k \cdot 2^n$, $b < 2^n$, then $a \oplus b > k \cdot 2^n$.

(3) For a node A and a distance D, there is only one node X, such that $D = \overline{AX}$. Therefore, if a bucket's distance range is D_1 to D_2, where $D_2 - D_1 = M$, then there are exactly M IDs in the bucket.

Now, let us prove that for every point P such that $\overline{BP} < 2^n$, the distance $\overline{AP}$ satisfies the following:

$$\overline{AP} < (j+1) \cdot 2^n \tag{9.1}$$

For the demonstration, we will start with the following identity:

$$\overline{AP} = A \oplus P = A \oplus P \oplus B \oplus B = A \oplus B \oplus B \oplus P = \overline{AB} \oplus \overline{BP} \tag{9.2}$$

Now, we know that $\overline{AB} < (j+1) \cdot 2^n$, since B is contained in A's bucket. Since $\overline{BP} < 2^n$, we can apply *(1)* with $k = j + 1$ to $\overline{AB} \oplus \overline{BP}$ and Equation 9.1 is reached.

Let us now prove that, for every P such that $\overline{BP} < 2^n$, it holds that:

$$\overline{AP} \geq j \cdot 2^n \tag{9.3}$$

We know that $\overline{AP} = \overline{AB} \oplus \overline{BP}$ and that $\overline{AB} \geq j \cdot 2^n$, since B is contained in A's bucket. Given that $\overline{BP} < 2^n$, we can apply *(2)* with $k = j$ to $\overline{AB} \oplus \overline{BP}$ and reach Equation 9.3.

With Equations 9.1 and 9.3, we have proved that $\overline{AP}$ falls in A's bucket, for any P such that $\overline{BP} < 2^n$. Now, from *(3)*, we know that the number of points associated with A's bucket is $(j+1) \cdot 2^n - j \cdot 2^n = 2^n$. The same reasoning applies to the range $\overline{BP} = [0, 2^n)$ and so it also contains 2^n points. It follows that the points in both ranges are the same.

9.4. Fighting Churn

Churn and node failures are the main issues that broadcast algorithms have to deal with. Both factors cause loss of messages, either because they are sent to a stale contact, they are lost by the network or because a contact fails to forward them. In this section, we present a theoretical model to calculate the expected coverage for broadcasts in Kademlia under failure conditions. Then, we introduce techniques implemented on top of PB and BB algorithms in order to overcome these problems and increase coverage. In general, the use of these methods trades better coverage for an increased latency or a higher number of messages.

9.4.1. Expected Coverage Under Failure Conditions

9.4.1.1. Probabilistic Model

Czirkos *et al.* developed a model to calculate the expected coverage of a broadcast in a Kademlia network, depending on the rate of network failures in the system [102]. If P is the probability of correct delivery of a single message and D is the depth of the broadcast tree (dependent on the number of nodes in the network), the expected number of nodes receiving the broadcast is:

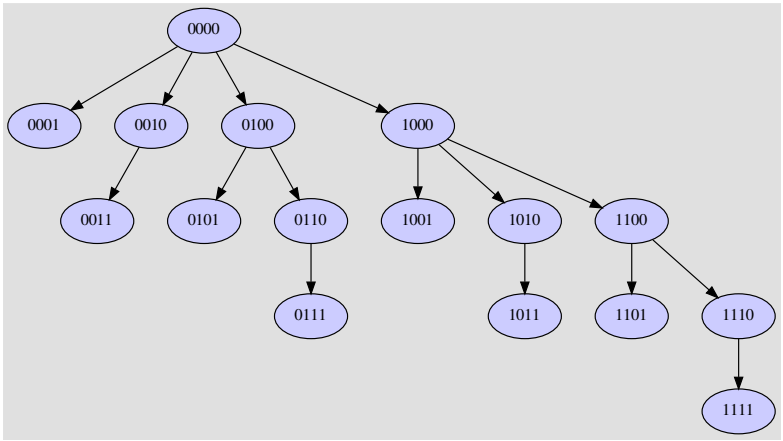
$$E = (1 + P)^D \tag{9.4}$$

This model however only applies to the binary version of Kademlia (b=1) and the tree created by the bucket-based algorithm. We have extended the analysis to partition-based algorithms and arbitrary b values by realizing that, in fact, the probability that a broadcast reaches a node only depends on the number of individual messages that must succeed to reach it. The nodes in the first level of the broadcast tree have a probability P of receiving the message, the nodes in second level have a probability $P \cdot P = P^2$—since two individual messages must succeed for them to be reached—and nodes in level i will be reached with probability P^i. The expected coverage for the

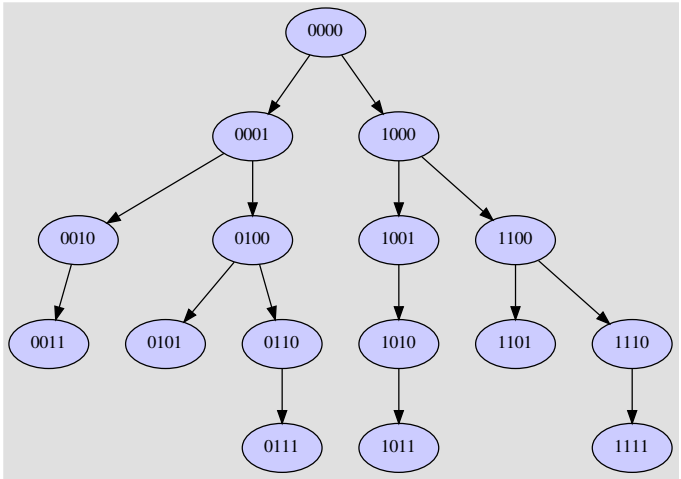
tree will be thus:

$$E = 1 + \sum_{i=1}^{D} n_i P^i \tag{9.5}$$

where n_i is the number of nodes per level in the tree (notice that we add one for the node initiating the broadcast, which we can consider to sit at level 0).



(a) Bucket-based tree



(b) Partition-based tree

Fig. 9.2: Broadcast trees for a network of 16 nodes.

We can thus build the broadcast tree for a given number of nodes, count

the number of nodes per level, add the probability that they are reached and thus determine the expected coverage of the broadcast. As an example, Figure 9.2 shows two possible broadcast trees for a network of 16 nodes. Figure 9.2a shows the tree created by the BB algorithm and Figure 9.2b shows the tree of a PB broadcast. The number of nodes per level at each one is $(1, 4, 6, 4, 1)$ and $(1, 2, 4, 6, 3)$, respectively. This means that the expected coverage is $1 + 4p + 6p^2 + 4p^3 + p^4$, for BB, and $1 + 2p + 4p^2 + 6p^3 + 3p^4$, for PB. Notice that while the PB algorithm creates a balanced tree, this causes more nodes to be moved to deeper levels in the tree and therefore the expected coverage is lower than in the BB case.

We must indicate here that in our implementation, we have modelled message errors as nodes failing to forward a message—rather than to receive it. Thus, nodes at level 1 always receive the message and nodes at level 2 receive the message with probability P. The resulting expected coverage will therefore be:

$$E = 1 + \sum_{i=1}^{D} n_i P^{i-1} \tag{9.6}$$

This coverage is somewhat higher than the one previously indicated. In what follows, we will use this formula when computing coverage to be able to compare with experimental results.

9.4.1.2. Coverage per Type of Broadcast

In order to apply the described probabilistic model to calculate the expected broadcast coverage, we have produced broadcast trees of different heights for BB configurations with b=1 and b=3, and also for PB setups producing the same tree depth (λ=2 and λ=8, respectively). Table 9.1 shows the number of nodes per level and the resulting expected coverage for the cases where there is a 5 % and 10 % error rate when forwarding a message. Configurations with higher number of contacts in the first steps (higher b or λ and BB rather than PB) achieve better coverage.

In order to check the validity of these estimations, we can compare the result for a network of 1,024 nodes with the actual values measured in a real Kademlia system with 1,000 nodes (experiments discussed in depth in Section 9.5). Notice that the model was developed for a fully populated network while in reality we have a few nodes scattered around a huge and almost empty key space. However, since node identifiers are generated with a hash function, they are uniformly distributed in the key space so the models should hold valid. Table 9.2 compares the expected coverage based on the average number of nodes per level found in the tests and the one experimentally measured). In every case, the coverage is calculated/measured for 5 % and 10 % failure rates. We can see that the results are similar to our theoretical expectations (1,024 nodes case, at Table 9.1).

Table 9.1: Theoretical coverage per type of broadcast.

Type	Nodes	Number of nodes per level	Reached	Cover (%)
BB b=1	16	1, 4, 6, 4, 1	15 / 14	94.8 / 89.8
	64	1, 6, 15, 20, 15, 6, 1	57 / 52	90.3 / 81.5
	256	1, 8, 28, 56, 70, 56, 28, 8, 1	220 / 188	85.9 / 73.7
	512	1, 9, 36, 84, 126, 126, 84, 36, 9, 1	429 / 358	83.8 / 70.0
	1024	1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1	836 / 681	81.7 / 66.5
PB λ=2 $b \in 1, 3$	16	1, 2, 4, 6, 3	14 / 13	92.4 / 85.3
	64	1, 2, 4, 8, 15, 22, 12	54 / 45	84.5 / 71.1
	256	1, 2, 4, 8, 16, 32, 60, 86, 47	196 / 149	76.7 / 58.3
	512	1, 2, 4, 8, 16, 32, 64, 120, 171, 94	373 / 269	72.9 / 52.6
	1024	1, 2, 4, 8, 16, 32, 64, 128, 239, 342, 188	709 / 485	69.3 / 47.5
BB b=3	16	1, 11, 4	15 / 15	98.8 / 97.5
	64	1, 19, 44	61 / 59	96.6 / 93.1
	256	1, 27, 148, 80	240 / 226	94.1 / 88.3
	512	1, 31, 224, 256	475 / 440	92.9 / 86.1
	1024	1, 35, 316, 608, 64	939 / 859	91.8 / 83.9
PB λ=8 b=3	16	1, 8, 7	15 / 15	97.8 / 95.6
	64	1, 8, 55	61 / 58	95.7 / 91.4
	256	1, 8, 64, 183	234 / 214	91.8 / 83.9
	512	1, 8, 64, 439	465 / 422	91.0 / 82.5
	1024	1, 8, 64, 504, 447	907 / 800	88.7 / 78.2

Table 9.2: Nodes per level and coverage for 1,000 nodes.

Type	Average nodes per level	Coverage (%) Levels	Experim.
BB b=1	1, 10, 48, 122, 212, 247, 194, 108, 44, 12, 1	81.9/66.9	81.1/64.3
PB λ=2	1, 2, 4, 8, 16, 32, 64, 128, 238, 290, 171, 43, 3	69.1/47.2	68.1/47.6
BB b=3	1, 22, 161, 464, 326, 27	89.5/79.8	90.1/80.3
PB λ=8	1, 8, 64, 434, 454, 39	88.2/77.5	89.3/76.6

9.4.2. Empty Regions Re-assignment

By default, broadcasts ignore regions without contacts. In stable conditions, this is safe, but in the presence of churn, routing information can be momentarily inaccurate and new nodes may be unknown. The ERR (*Empty Regions Re-assignment*) routine lets empty regions be handled by contacts in close regions, in the hope that they may know about nodes still ignored by the current handler.

The application of this algorithm does not increase latency or number of messages so it can be safely added to the bare protocols. However, under severe churn or failure conditions, it offers limited node coverage gain. This can be seen in Figure 9.3, which shows the difference in coverage between protocols with and without the application of the ERR technique. When only churn is considered, ERR provides some coverage improvement, however, when non detectable failures appear, ERR does not seem to offer a significant improvement over their non-ERR counterparts (the dispersion is too high to obtain clear results).

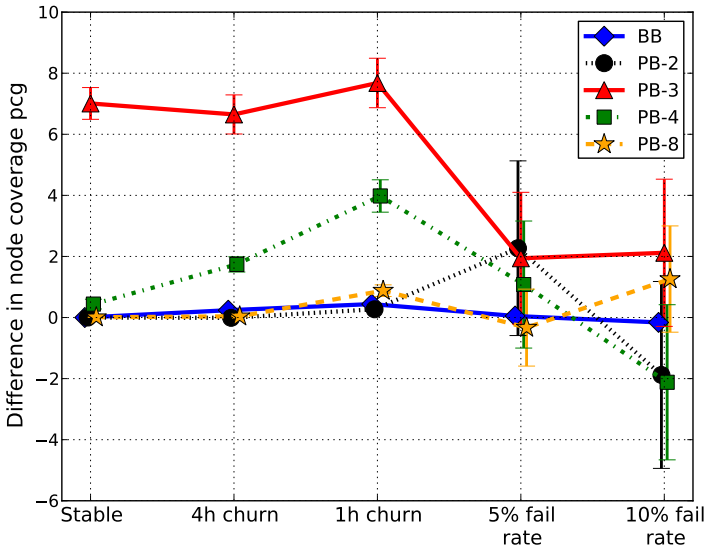


Fig. 9.3: Gain in node coverage when applying ERR.

We may regard ERR as an integrated improvement of the core protocols rather than an additional technique. Therefore, we will not study it separately in Section 9.5.

9.4.3. Redundancy

One of the advantages of Kademlia is that it can send several parallel messages to a given region. If one message is lost, some other may reach its

destination. The advantage of this technique is that it adds no latency since parallel messages are sent almost simultaneously. The drawback is obviously an increase in the number of sent messages.

Notice that when redundancy is used, it may occur that a node receives the same message several times. In order to ensure that nodes process messages (pass them to upper layers) only once, a unique identifier is given to each one. In addition, a policy must be defined so that nodes forward only the first time they receive a message—F1 (*Forward one*)—or, otherwise, forward it every time—FA (*Forward all*). In regard to the replication of messages, there are also different possibilities: replicate only at the broadcast initiator—R1 (*Replicate one*)—or at every step, i.e., forward duplicated messages to every handled subregion—RA (*Replicate all*).

Considering all this, we have four possible configurations: R1-F1 (*Replicate one/Forward one*), R1-FA (*Replicate one/Forward all*), RA-F1 (*Replicate all/Forward one*) and RA-FA (*Replicate all/Forward all*) (notice that Czirkos only considers RA-F1). We can immediately discard the RA-FA option because it would lead to an explosion of messages on the network. However, although perhaps not so intuitive, we can also discard the first combination, R1-F1, because it causes a coverage reduction instead of a gain. Indeed, even in the absence of churn or failures, a node may be missed because redundant messages are discarded. We will call this effect *paths interference*.

An example of the effect is shown in Figure 9.4, which depicts a broadcast with a redundancy factor of 2, reaching a 16-node subregion. In the matrix, rows represent nodes with different IDs, columns indicate their state after each forwarding step and messages are drawn as arrows. The ID space is not fully populated and empty positions are marked with dashed lines. Present nodes are filled with a light colour when they have not received the broadcast and with a dark one when they have. In the example, two identical messages arrive at nodes 0010 and 0111, which forward it to the upper and lower halves of their region. Looking at the upper part, we see that both nodes contact the same destination (0000). This node will discard one of the messages and forward the other one. Unfortunately, in the second step, node 0000 randomly chooses node 0111, which has already processed the broadcast and will thus discard it this time. Node 0000 does not have any other contact in the upper part of its handled region so it does not send any other message. Nodes 0100 and 0101 have missed the broadcast.

This effect has been confirmed experimentally and that is why the R1-F1 policy for redundancy configuration has not been considered for the evaluation tests in Section 9.5. Actually, the *paths interference* also affects the RA-F1 configuration in some cases but, as we will see, the replication at every step is, in general, enough to compensate for it.

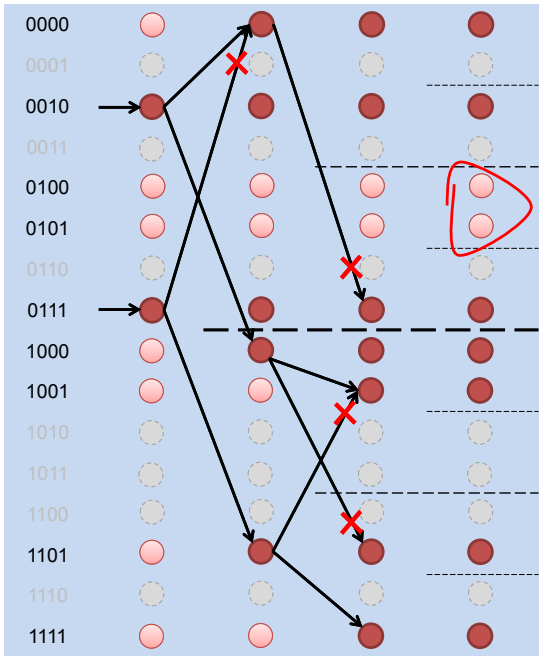


Fig. 9.4: Paths interference when using redundancy with R1-F1 policy.

9.4.4. Direct ACKs and Resubmissions

In our bare implementation of the protocols, a node forwards a broadcasting message and forgets about it. However, in order to fight unexpected failures, we have also enhanced the protocols with the sending of an ACK as soon as a node has correctly processed and forwarded a received message. If the ACK is not received, the sender will resubmit to a different contact in the region at hand.

Direct ACKs add latency for each resubmission but not as much as preceding ACK strategies. Our method requires a timeout adjusted to the reply of the immediate recipient. In previous cases, the timeout had to be big enough for the message to propagate through the whole subtree below the node [99].

If only ACKs and resubmissions are applied, nodes should only receive a message once, since no parallel paths are used. Only when a node fails to be contacted or to forward a message, another contact is tried out. In any case, if, for some reason (such as ACK loss), redundant paths were in place, F1 would be the most appropriate policy for this case. The reason is that instead of forwarding the same message again, it is better to discard it and let the sender look for another contact to propagate the broadcast.

Indeed, if we apply this technique to the same 16-node region we saw earlier (with two identical messages arriving at the area), we find that now all

nodes are reached. This is shown by Figure 9.5, where resubmitted messages are represented by dashed arrows. The key point is the resubmission of node 0111 after its message to node 0000 was not acknowledged (because it was a duplicate). This time, node 0100 is contacted and, in turn, it informs 0101. In this scenario with duplicated messages, ACKs cause some useless resubmissions but they are discarded and perfect coverage is achieved after a few extra steps (one, in this case).

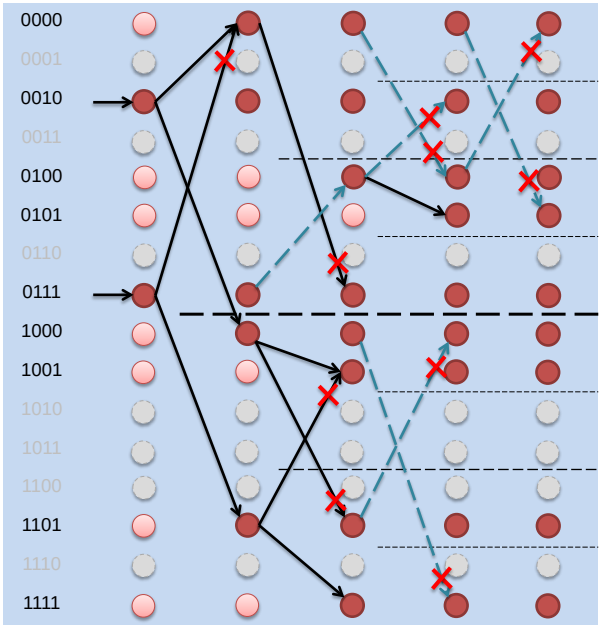


Fig. 9.5: Duplicated messages when using ACK and resubmissions.

9.4.5. Other Churn Fighting Techniques

Redundancy and resubmissions are simple yet powerful techniques to fight churn and malfunctioning nodes. However, as described in Section 9.5, they also present some limitations. In order to assess if there are ways to overcome these, we have implemented the possibility to apply both techniques at the same time. In particular, we have evaluated the R1-FA configuration with the addition of resubmissions.

In order to assess a totally different approach, we have also implemented a simple flooding technique that complements the structured broadcast algorithm. The strategy is to let every node forward the received message to a number of random contacts in addition to those indicated by the standard protocol. In this case, one must be very careful because the use of the *forward-all* policy results in the number of messages rising in every step and this may lead to a congestion of the system. If *forward-one* is applied

instead, the coverage results are very modest. As per our experiments, the best outcome is achieved when a *forward-some* rule is used; i.e., when forwarding a few (e.g., 3) of the duplicate messages that are received. This is the configuration shown in Section 9.5.

9.5. Evaluation

9.5.1. Testbed and Setup

In order to test the performance of the described protocols and techniques, we have built a real Kademlia system with 1,000 peers running on 50 physical nodes in a parallel cluster facility at the CIEMAT institute. A master process instructs the nodes to start new peers, initiate a broadcast or make peers leave the system. More than 12 000 CPU hours have been necessary to gather the results here presented. Once the system is up and stable (peers have joined the network and filled their routing tables), the master sets the desired churn conditions by adding and removing peers at the appropriate frequency. When a peer leaves the system, it does not come back to life. No peer leaves the system during the propagation of the broadcast, which anyhow completes quickly given that all peers run in a single multiprocessor computer. More details of the architecture and flow of operations testbed are given in Section A.5 of Appendix A.

Three situations are studied: completely stable system (no churn), average node lifetime of 4 hours (joins and leaves every 14.4 seconds) or average node lifetime of 1 hour (joins and leaves every 3.6 seconds). These numbers do not reflect real patterns of nodes in well known Kademlia networks; they instead try to match (or be under) typical worker nodes lifetimes in scientific grid sites. This is the environment in which we have used Kademlia broadcast, in the algorithms of the TQ discussed introduced in Chapter 10. In any case, the important point here is to assess the robustness of the protocols and churn fighting techniques when routing contacts become stale.

We have also considered nodes failing to forward received messages. This scenario is worse than pure churn because, after successful delivery, the sender will in principle not look for alternative routes (unless ACKs and resubmissions are applied). The cases where 5 % or 10 % of the nodes are unresponsive have been tested.

The master asks random peers to initiate a broadcast every 15 seconds during 10 minutes. The process is repeated 3 times. All the metrics described in Section 4.5 have been measured for the bucket-based algorithm (BB) and for the partition-based protocol with four different values of λ (PB-2, PB-3, PB-4, PB-8). In every case, b=3 was used. We have set different churn and failure conditions and studied the effects of using redundant paths (R1-FA, RA-F1), direct ACKs and resubmissions and flooding.

9.5.2. Coverage under Different Conditions

Figure 9.6 shows the coverage achieved by the different broadcast protocols under different churn and failure rate conditions. The graph shows average values and standard deviation values as error bars. The legend is shared by all subplots.

Figure 9.6a compares the behavior of the bare protocols. The first three measured cases (stable and churn conditions) show that the bucket-based and partition-based algorithms with λ=2 or λ=8 achieve full coverage of the nodes in the absence of churn and coverage over 98 % even when churn is present. With λ=4, the coverage is good in stable conditions but behaves worst with churn. With λ=3, the results are worst in all cases. As discussed in section 9.2.1, this was expected since only for λ=2 and λ=8 can the forwarding regions and subregions be exactly mapped to buckets.

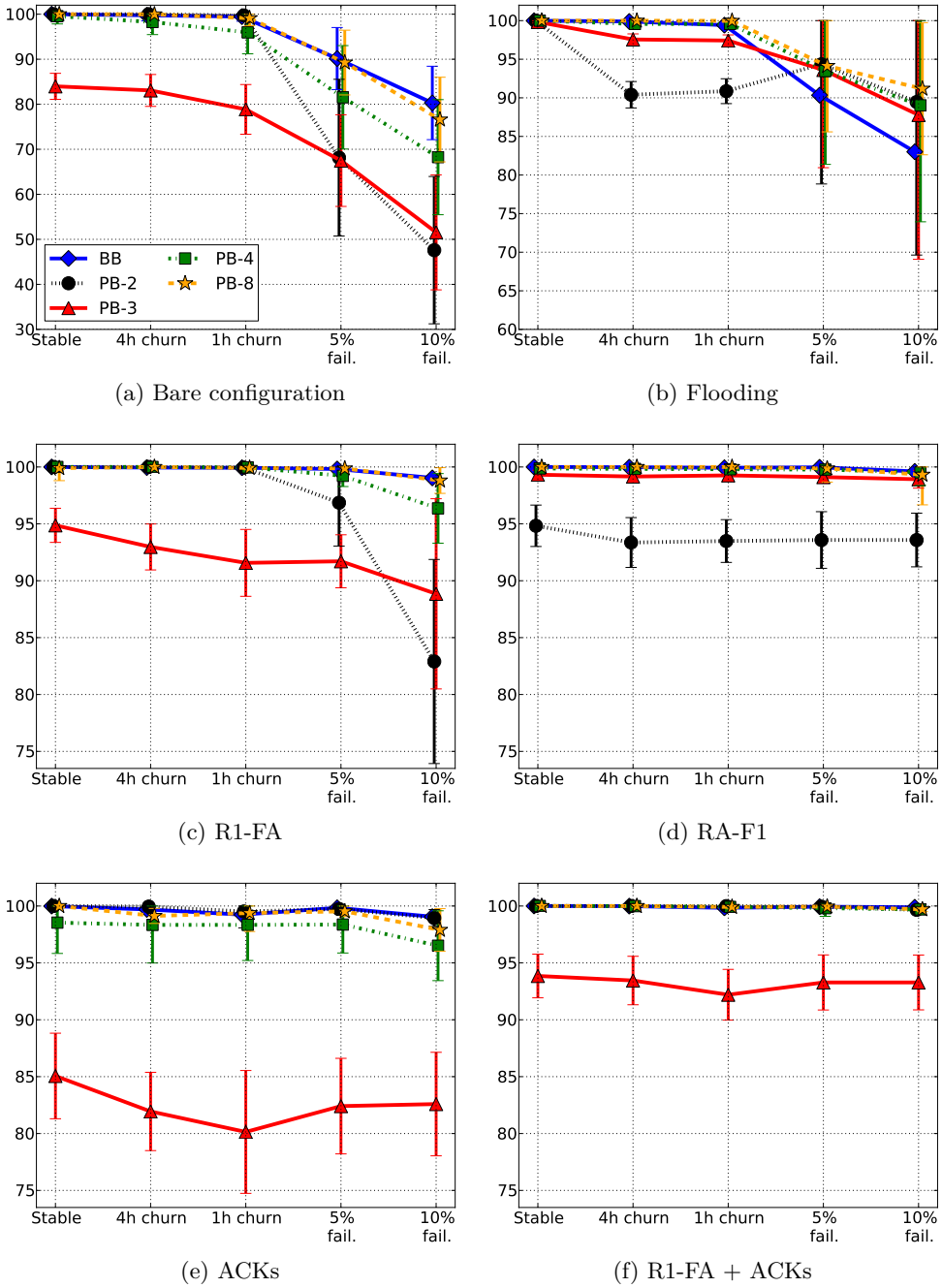
When random failures are applied (4th and 5th cases), all algorithms show worse coverage and much greater dispersion. Configurations with narrower regions behave better (PB-8 and, even better, BB) while partition-based with λ=2 is the most fragile configuration. In this case, a failing node causes a whole branch of the binary tree to miss the broadcast. This is well in line with the predictions in Section 9.4.1

9.5.2.1. Flooding

Figure 9.6b shows the coverage results obtained when the simple flooding technique is applied. Even with the best forwarding rule (*forward-some*), the outcome is not very satisfactory. Especially for the configurations with non-zero failure rate, the coverage is poor and the dispersion is huge. In a structured network like Kademlia, it is more efficient to add redundancy or resubmissions than random flooding.

9.5.2.2. Redundant Paths

In order to reduce the number of possible configurations to test, we have established a fixed replication factor of 3. This is the typical redundant factor in Kademlia but others could be used. We have tested two different configurations: R1-FA and RA-F1. Figure 9.6c shows the node coverage with the R1-FA configuration and Figure 9.6d shows the RA-F1 case. We can see that the use of redundant paths helps to greatly improve coverage in all cases. In the R1-FA case, results are worse when PB-3 is used and for the cases where random failures are present. The RA-F1 configuration achieves better results in general (and mostly almost perfect) except for the PB-2 protocol, which suffers from the paths interference (mentioned in Section 9.4.3) more than the other configurations. This is seen even in stable conditions.



(a) Bare configuration

(b) Flooding

(c) R1-FA

(d) RA-F1

(e) ACKs

(f) R1-FA + ACKs

Fig. 9.6: Node coverage (%) by protocol with different configurations.

9.5.2.3. ACKs and Resubmissions

We have also implemented the resubmission of a message when an ACK was not received for it. Figure 9.6e shows the coverage obtained with this new policy. We see that there is an increased coverage in all cases but there is some difference with the results observed for redundancy. Firstly, ACKs fail to improve results for PB-3 as much as redundant paths do. Since these failures are due to incorrect mapping of regions to buckets (and not failures or churn), some contacts are simply missed and resubmissions do not help. Redundant paths however may find their way to them. On the contrary, ACKs achieve better results for PB-2 with non-zero failure rates. In such cases, a branch is missed if all redundant paths come to a failing node even if other contacts are working fine. ACKs make it possible to resubmit to those. In other words, this technique is not affected by paths interference.

9.5.2.4. ACKs and Redundancy Combination

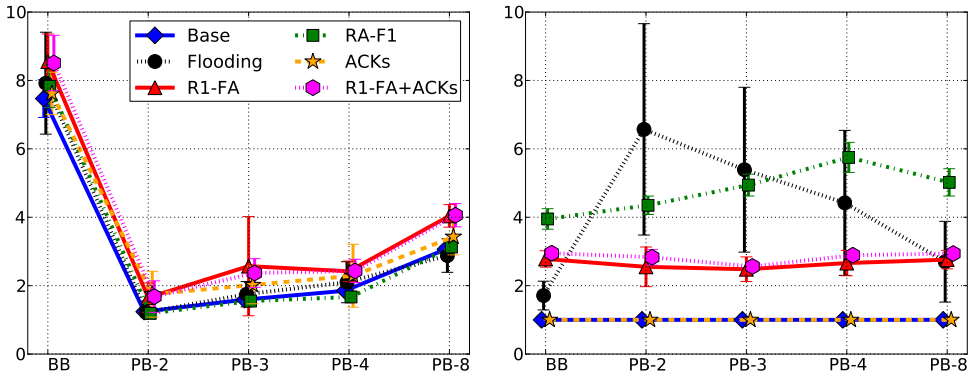
As we have seen, ACKs and redundancy achieve very good results but the first obtains better coverage for the PB-3 protocol while the latter is better for PB-2. We have also combined both techniques (R1-FA and resubmissions) and used them at the same time. The results are summarized in Figure 9.6f. We observe that coverage has been improved for all combinations (and PB-2 in particular). Still, PB-3 is somewhat worse than the others, but it is clearly better than with ACKs alone. In any case, based on all the results we have seen, PB-3 does not seem like a very sensible protocol choice in the first place.

9.5.3. Other Metrics

Apart from coverage, there are other interesting metrics to analyze. Figure 9.7 shows the imbalance factor, messages to nodes ratio and tree depth of every protocol and churn fighting technique configuration.

9.5.3.1. Imbalance Factor

Figure 9.7a shows the results regarding the imbalance factor. As expected, the bucket-based algorithm presents a higher value than the partition-based configurations. Among those, the imbalance factor increases slightly with the value of λ. This factor does not seem influenced by redundant paths with F1 policy and only slightly by the use of ACKs. When a FA policy is applied, the factor increases a bit more, but we do not think this can be considered significant (clearly minor compared to the use of a BB protocol).



(a) Imbalance factor

(b) Messages to nodes ratio

(c) Tree depth

(d) Steps to reach 90 % of nodes

Fig. 9.7: Other metrics for different protocols and configurations.

9.5.3.2. Messages to Nodes Ratio

Figure 9.7b shows that all considered protocols are very close to the ideal value of one message per node when no redundancy is used (ACKs alone do not have perceptible influence here). When redundant paths are added to the picture, the ratio increases to a value close to the redundancy factor (3, in this case) if the replication policy is of type R1-FA. This was to be expected. If resubmissions are also used, a slightly higher value is obtained, due to additional resubmission messages being sent.

If replication in every step is used (RA-F1), the number of messages increases significantly. Given that many redundant messages are discarded, it is quite difficult to estimate this value theoretically but the plot shows that, for 1,000 nodes and this replication factor, the final ratio ranges from 4 to 6 (doubling that of R1-FA). Finally, the flooding configuration also produces many messages since it causes the submission of new random messages in every forwarding step. The values obtained here are only limited by the applied *forward-some* policy. If *forward-all* had been used, the number of messages would have risen exponentially.

9.5.3.3. Tree Depth (Latency)

Figure 9.7c shows the tree depth information. The results make it clear that the algorithms that use a higher number of forwarding regions achieve completion of the broadcast in fewer steps. In this respect, BB and PB-8 obtain the minimum tree depth.

Simple redundancy (R1-FA) and flooding do not have a big impact in the depth of the tree. However, resubmissions (with or without added redundancy) do increase it. The reason is clear: to detect a failure, a node has to wait for a timeout to occur, before resubmitting. For our tests, we have modeled this by increasing the step number of a resubmission in 4 (corresponding to a timeout of twice the round-trip-time between nodes).

Interestingly, in the RA-F1 configuration the number of steps is reduced below that produced by the bare configurations. The continuous replication increases the number of nodes reached in every step of the broadcast and the whole network is covered earlier. The R1-FA policy also increases the number of nodes in the first levels of the broadcast trees, thus reducing the count on the last steps of the process. This is shown in Table 9.3, which compares the average number of nodes per level of each configuration (taking the PB-2 protocol as an example). We see that the number of nodes in the last levels of the R1-FA and flooding configurations have fewer elements than the cases with no redundancy. It might happen that in some cases these techniques also reduce the tree depth slightly, but not so much as RA-F1.

It is also important to notice that the previous depth values reflect the global latency, i.e., the delay for the final nodes reached by the broadcast.

Table 9.3: Nodes per level by configuration for PB-2 and 1,000 nodes.

Configuration	**Average nodes per level**													
Nb. level	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Bare protoc.	1	2	4	8	16	32	64	128	238	290	171	43	3	
Flooding	1	2	6	17	45	98	185	259	236	114	30	6	1	
R1-FA	1	6	10	15	22	41	77	134	235	279	150	28	2	
RA-F1	1	6	28	88	127	166	202	188	128	13				
ACKs	1	2	4	8	16	32	64	128	236	295	170	39	4	1
R1-FA+ACK	1	6	10	15	21	41	79	133	235	272	153	32	1	

The nodes that miss the broadcast are not taken into account (otherwise we would have infinite depth in many cases). This is not really fair since the latency value of the configurations with best coverage corresponds to a higher number of nodes. This is illustrated by Figure 9.7d, which shows the number of steps required to reach 90 % of the nodes that get the broadcast. Since the last few nodes are the most difficult to reach, values are much more even in this case (except for RA-F1 which still achieves a significant improvement over all the other configurations).

9.5.4. Summary of Algorithms Evaluation

The results of our experiments show that in stable conditions BB, PB-2 and PB-8 achieve perfect coverage of the network. Even in the presence of churn, the coverage is very good. As expected, the use of partition-based algorithms with other values of λ (such as 3 or 4) produces worse results. PB-2 also achieves the lowest imbalance factor but it produces significantly deeper broadcast trees (trading load balancing for latency). In any case, the imbalance factor of PB-8 is only marginally worse than that of PB-2 while it achieves minimum tree depth, so it seems a good choice for most situations. As for messages to nodes ratio, unless redundancy or flooding are used, all protocols achieve the optimum value, 1.

When random silent failures are added to the system, things look different. PB-2 algorithm appears as the most fragile option because the average number of messages required to reach a node increases. As predicted by the probabilistic model, configurations with higher number of nodes on the top levels of the broadcast tree offer the best coverage, namely, BB and, next, PB-8. The improvement achieved by the first one comes at the cost of a higher imbalance number.

However, under message failure conditions, the use of redundant paths and direct ACKs becomes a necessity. Both cause a dramatic increase in node coverage: best results are obtained with RA-F1, followed by those of ACKs and resubmissions. The use of ACKs increases latency in the propagation of the broadcast while redundant paths increase messages to nodes

ratio instead. If the RA-F1 configuration is used, the tree depth is reduced below the one achieved by the bare protocols, at the cost of an even higher number of messages. When both redundancy and ACKs are used at the same time, the coverage is excellent (even better than with RA-F1), the ratio of messages behaves like with R1-FA and the tree depth is similar to the one obtained with ACKs alone.

Therefore, depending on application requirements (in terms of latency) and network characteristics (round-trip-time between nodes, average load of the nodes), one method or another should be chosen. All things considered, if load is not a problem or if latency is critical, the RA-F1 configuration would be the best choice. But if our priority is to reduce the produced number of messages, then ACKs alone seem a very interesting choice. Between these two options, the combination of R1-FA and ACKs increases the ratio of messages to nodes less than RA-F1 and achieves the best coverage, with a somewhat higher latency. As a final remark, the possibility of using a simple flooding technique does not offer very interesting results. It produces a very significant increase in the number of sent messages but the achieved coverage enhancement is far from impressive.

WRAP-UP: *We have analytically discussed and experimentally confirmed that full coverage broadcasting, with optimum ratio of messages to number of nodes, is achievable in Kademlia systems by using only the DHT contact tables. This can be done by using our proposed bucket-based algorithm or by applying a partition-based protocol, but only if the splitting degree (λ) is compatible with the buckets configured in Kademlia.*

The node coverage results are also very good when churn conditions are present and can be improved even further by applying direct ACK and resubmissions or by profiting from Kademlia's characteristics and using simultaneous parallel messages. They carry the cost of an increase in latency (ACKs) or ratio of messages to nodes (redundancy). Overall best coverage results are obtained with the combination of both techniques at the same time but care must be taken to choose the appropriate forwarding policy since duplicate messages can interfere with broadcast propagation.

Chapter 10

Distributed Data Caching and Job Matching

Chapter 7 introduced the Task Queue architecture, a new late-binding overlay that incorporated a data cache to reduce the I/O pressure on a site's SE and used intelligent scheduling to make tasks run on the nodes holding their required input data. The described architecture however proved to be very demanding on the central server, the TQ. In order to maximize the cache hit ratio, the TQ must keep track of every data product in the nodes and, whenever a pilot requests workload, all queued tasks must be examined. We have observed that this can lead to severe matching delays when the scale of the system (number of pilots and tasks) increases.

In order to overcome this problem, we have developed a new system that offers several enhancements to the original TQ architecture. In the new system, pilot nodes arrange themselves into a Kademlia network within a site's LAN. Pilots share the DHT and use it as common key-value store (in particular, for the location of cached files) as well as to locate and reach other nodes. In this way, pilots are able to retrieve their required input files from other peer nodes, greatly increasing the effective hit ratio of the cache. Moreover, the new inter-pilot communication capabilities have made it possible to implement a distributed matching algorithm, by which pilots in a site collectively evaluate, rank and assign computational tasks. The TQ, freed from tracking cache files and performing the task matching, becomes a much more scalable component and the pilots gain some autonomy from the central server. The latter also means fewer wide area network interactions, which becomes important when latencies are high.

The description and evaluation of the new architecture is the focus of this chapter. Section 10.1 discusses the characteristics and enhancements of the implemented Kademlia DHT. Section 10.2 comments on the pilot-based distributed data cache and Section 10.3 analyzes the new task matching procedure and its impact on the achievable global rank. Finally, Section 10.4

and 10.5 present the results of a long series of tests assessing the performance and main characteristics of the whole system.

10.1. Custom Kademlia Implementation

The architecture of the new TQ requires the use of a modified DHT for both the shared cache and the distributed task matching procedure. Our implementation of the Kademlia protocol introduces a few changes to the original specification. Firstly, when a pilot needs to leave the system, it sends a `Leave` message to its known contacts, so that they update their routing tables, and it republishes its key-value pairs so that the storage redundancy is kept. This contributes to a more proactive adaptation to node departures in contrast to the expiration-based behaviour of pure Kademlia. Secondly, if a `Store` message is received for an already known key, the new value is appended to the existing one. This makes it possible to associate multiple locations (replicas) to a single file, increasing the chances of later retrieval. Finally, while Kademlia authors propose to expire newest contacts first, based on the observation that typically oldest DHT contacts remain alive longer, we apply a simple least-recently-seen expiration policy. We do so because grid pilots are expected to have a fairly constant lifetime, determined by batch system limits.

The intra-site matching and ranking process depends on the dissemination of the list of runnable tasks from the master to the workers (`TaskList` message) and the collection of results by the master. This required a major enhancement of the Kademlia protocol to add support for broadcast and aggregation capabilities, which resulted in an in-depth study of the broadcast problem, described in Chapter 9. In the current implementation, each node divides the space it must broadcast to in eight regions and forwards the message at hand to one of its contacts in each region. This means that, no matter the size of the network, the master will never need to send messages to (or receive from) more than eight workers. For those broadcasts that require an aggregated reply, each receiving node keeps a table of expected replies and a pointer to its *parent node* (the one sending the message to it). When all the replies are received or after some timeout, the aggregated result is sent back to the parent. Late messages are simply discarded but this is not too serious: a pilot not getting a task in a given round just needs to wait for the next call to come.

10.2. Distributed Data Caching

We have used the Kademlia DHT to build a distributed data cache, with the following goals:

- Relieve the Task Queue of the duty of keeping track of all the data files in the pilots cache.
- Make it possible for pilots to locate and fetch files from peer nodes when required (increase cache hit ratio).

When a pilot job is started on a WN, it first contacts the TQ to register itself. At that point, the TQ assigns a random Kademlia identifier to it and provides it with a list of contacts at its site. With this information, the pilot can initiate the procedure to join the corresponding network by reaching those contacts and filling some entries of its DHT routing tables. Correspondingly, it will be added to the tables of its peer nodes as necessary. The node is now part of the Kademlia network and can ask for tasks to run.

The sharing of files via the DHT works as follows: When a new file is produced, the pilot stores a key-value pair in the DHT, where the key is the hash of the file name (or unique identifier) and the value is its location. This information is replicated on a number of nodes (typically, three) that are close to the key according to Kademlia metrics. When a pilot is assigned a task for execution, it examines its data dependencies. Every required file is first looked for in the local cache and, if not there, a DHT finding procedure is initiated with the hash of the file name as argument. Once the location of a replica is retrieved, the pilot fetches the file and adds it to its local cache. For this purpose, every pilot runs a simple file server accepting requests from other peers. All this procedure takes place before the unmodified real job is started. At execution time, the job behaves as usual: it reads input data from local cache or, upon failure, falls back to the local SE.

10.3. Distributed Job Matching

As discussed in Section 8.2, the heaviest duty performed by central task queues, and in particular by our original TQ, is the assignment of real jobs to pilots. The TQ can become a bottleneck for the whole scheduling system. That is why we have designed a new DHT-based task assignment architecture, with the following aims:

- Reduce load on the TQ; avoid it becoming a bottleneck.
- Reduce the interactions of the pilots with the TQ, increasing their autonomy.

In our new architecture, there is a representative per site, called master pilot or, simply, *master*. Since every pilot needs to register with the TQ on wake-up, the TQ can assign the master role to the site's first registered pilot and inform the following ones about it. When the master pilot dies, a simple

election process is carried out to choose the pilot with lowest identifier and the TQ is informed about the result.

The master contacts the TQ and requests tasks for its site. The TQ only needs to match those tasks with no data dependency (able to run on any site) or depending on data held at the site's SE. The description and requirements of all the matching tasks are passed to the master, which broadcasts the information to the rest of the site pilots (*workers*). The workers filter and rank the tasks and send the results back to the master, which assigns a task to each pilot, trying to maximize the sum of individual ranks, and informs both the TQ and the workers about the mapping. The pilots proceed to download the task assigned to them, fetch the necessary input files and run it. When the task is finished, the pilots inform the master about it and upload the resulting report. Figure 10.1 shows a simplified diagram of the interactions among the different components of the system.

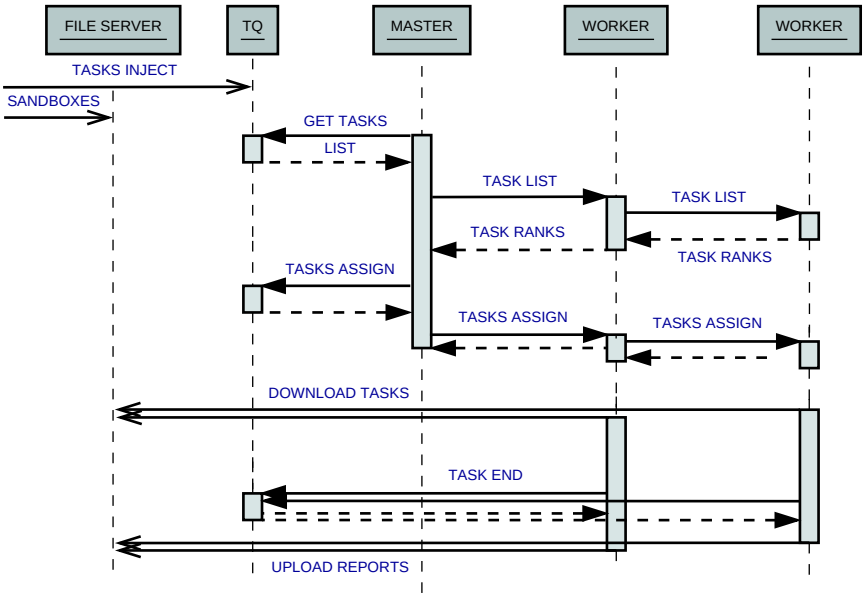


Figure 10.1: Simplified diagram of the distributed scheduling process.

From this description, we can see that the duties of the TQ are much lighter now. Firstly, it only receives task requests from one node per site. Secondly, these require a simple (bulk) selection based on a single criterion (site) and not a complex matching process. Furthermore, since the master node is in permanent contact with the workers, it can also inform about non-responding or dead nodes and therefore periodic heartbeat messages from workers are not needed anymore (just as the DHT meant the removal of the file tracking reports). At this moment, the only worker messages requiring TQ processing occur at registration, shutdown and task completion times. Not only have we greatly reduced the number of requests the TQ must serve

but we have also eliminated those involving costly computation. Moreover, it is also possible to avoid the worker's task end messages by letting the master inform about task completions in bulk. This has been actually implemented in a further development of the system and its impact is studied in Section 10.4.4. Finally, let us note that although the workers need to download real jobs and upload completion reports, these interactions entail no logical processing at the TQ other than the pure file serving/storing. In other words, the TQ does not perform any state change so these duties can be just performed by a separate file server.

10.3.1. Task Matching and Ranking

Even if with the new architecture pilots can fetch required files from peer nodes, it is still better for tasks to land on pilots holding the files they need, since reading from the local cache on disk should be cheaper than through the network. In more general terms, we set the following goal for our architecture:

- Produce a global rank at least as good as in the previous architecture.

The procedure to assign tasks to pilots is in practice not very different from that used in the centralized scheduling. As discussed in Section 7.3, the requirement and ranking expressions are the key to select the best task for each pilot. In fact, all the information the TQ uses for their evaluation is the description of the task and the characteristics of the pilot. By broadcasting task descriptions, individual pilots are able to match and rank the tasks with exactly the same results as when the TQ did it centrally. Since this is done on a per pilot basis, even complex matching and ranking functions can be used with no scalability problem.

Notice however that with the old scheduling procedure the TQ would match one pilot at a time while, in the new model, the master receives information from all the pilots in a site and then performs a collective assignment, aiming for the best global result. Indeed, with the received rank values per pilot, the master builds a rank matrix where the row and column of each value corresponds to the evaluated task and the reporting pilot respectively. Now the master needs to maximize the sum of a series of matrix elements with the constraint that only one element per row and per column can be chosen (a pilot can run only one task and a task can be run only by one pilot). This kind of problems are known to be NP-complete but we can choose to accept a suboptimal (but good enough) solution in order to reduce the resolution time. In particular, we have implemented an algorithm by which we iteratively choose the best possible rank and discard the selected pilot and task. More formally, the pseudo-code for this would be:

```
1.  m := Rank_Matrix.max()
```

```
2.  i, j := Rank_Matrix.find(m)   # Row & column for value 'm'

3.  Rank_Matrix.remove_row(i)
    Rank_Matrix.remove_col(j)

4.  If empty(Rank_Matrix):
      Then: Exit
      Else: GoTo 1
```

Since we are now manipulating a numerical matrix, the algorithm can be performed quickly and produces quite satisfactory results. As indicated, the previous centralized procedure matched one pilot at a time. We could describe it in similar terms to those shown above. In particular, we would just need to replace the first step of the preceding algorithm (*find the matrix maximum*) with one that finds the maximum value of the first row. I.e.:

```
1.  m := (Rank_Matrix.get_row(0)).max()
```

It is easy to see that the results obtained with the distributed architecture must be in general better than those of the centralized one. In order to confirm this, some tests have been performed and their results are presented in Section 10.4.3. In addition, this is also reflected in the evaluation of the cache hit ratios achieved with the TQ, as discussed in Section 10.4.2.

10.4. Evaluation

In order to evaluate the new architecture, a testbed capable of running more than 1,500 simultaneous pilots was built at the CIEMAT institute. Two sets of resources were used in the tests. The first one comprises around 600 slots from an Infiniband-based parallel cluster facility while the second one includes nodes of the institute's grid site, where around 950 jobs can be run in parallel. In our configuration, the two sets are viewed as different sites. More information about the testbed is given in Appendix A.6.

With the aim of comparing the scalability of the old and new architectures, the same workflow types used in Chapter 7 have been run. These are inspired by real data-driven WLCG workflows, which can benefit by the use of a data cache. Tasks are chained in two steps. Every task produces a single data file of 3 MB but those of the second step consume the data generated at the first one. The average task duration is 3.5 minutes. The workflow types are *serial chain* (each data file is read by a different task), *splitting* (each data file is read by two tasks) and *merging* (each task consumes two files). Unless otherwise indicated, all the described experiments show the results from three independent runs of each workflow type.

The independent variables considered in the experiments are testbed size and architecture. The size is defined by the number of tasks and pilots.

The values used for the tasks are 300, 1.5k, 5k and 10k, and, for the pilots (including the two resource sets), approximately 100, 500, 1.5k and 1.5k, respectively. As for the architecture, there are three different configurations. The first one (*pre-DHT*) was presented in Chapter 7: the central TQ keeps track of all data files and also performs the task matching. The second one (*centralized*) represents an intermediate stage, in which the DHT has been introduced and file location is tracked by the pilots, but matching is still centralized on the TQ. Finally, our newly proposed system is labeled *distributed*. In this case, files are tracked via the DHT and task matching is performed by the pilots themselves using the new distributed algorithm.

10.4.1. Pressure on Task Queue and Scheduling Overhead

The first set of results addresses the scalability of the three architectures under study. The presented measurements include the number of requests served by the TQ, the scheduling overhead and the total workflow time, for each of the configurations and testbed sizes. For all presented plots, the error bars represent the standard deviation of the mean.

Figure 10.2 shows the amount of requests received by the Task Queue during a workflow run. The value increases almost linearly with the number of tasks in the workflow. In the pre-DHT configuration, each task causes around 12 TQ requests. Pilots contact the TQ for both task retrieval and file tracking. When the DHT is introduced but the task matching is kept centralized, the number of requests decreases moderately (to 9 per task) because no messages are sent to track file creation/deletion. Finally, when the new distributed matching is used, the master pilot takes care of task matching and heartbeat interactions and thus the number of requests decreases to around 5 per task. Remember that, with this architecture, not only are there fewer interactions, but these are also lighter.

Relieving the TQ of the costly processing duties is the key to make it scalable. Figure 10.3 displays the delay between the end of a task in a pilot and the start of the next one. Since, in our tests, tasks are always ready to be served to pilots (or otherwise there are no more to be run), this delay is a measure of the scheduling overhead. We find that both the pre-DHT and the centralized architectures behave nicely for small and medium scales (around 5 seconds of overhead) but show worse performance (and much higher dispersion) when the number of pilots and tasks increase. The reason is clear: the TQ cannot keep pace with the increasing request rate (more pilots), especially because each request demands heavier processing (more queued tasks to review).

The distributed architecture however scales nicely and offers almost constant inter-tasks delays. For smaller testbeds, this interval is somewhat longer than in previous setups but remains low even for the largest testbed. In fact, the delay is consistent with the period between `TaskList` messages

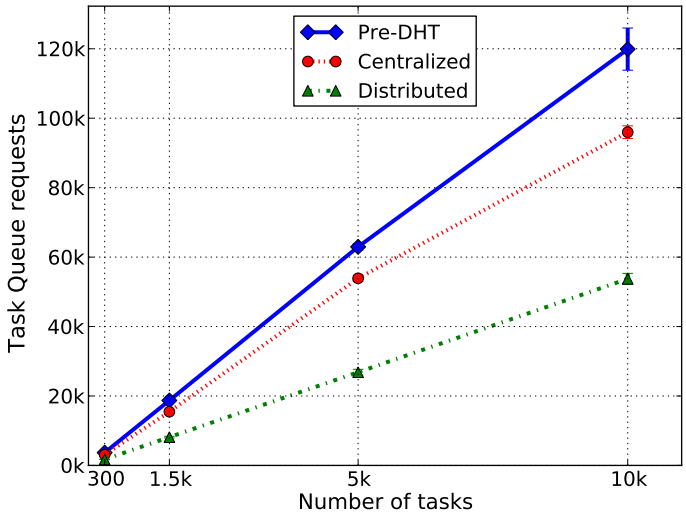


Figure 10.2: Number of TQ requests per architecture and testbed size.

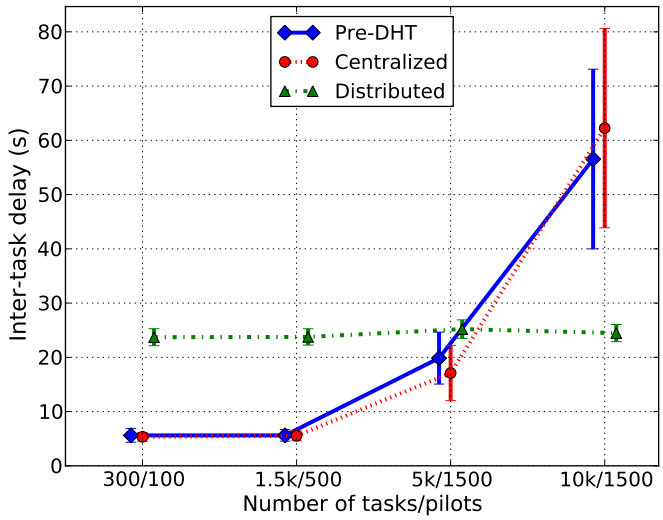


Figure 10.3: Inter-tasks delay per architecture and testbed size.

sent by the master (20 s). Even if this number is significant for our tests, where the average task duration is 3.3 minutes, it would be negligible for tasks in the order of hours (as expected in WLCG). Certainly, an increased task length would also result in a reduced request rate, thus alleviating the problems of centralized matching. Notice however that, first, this would not reduce the long queue of tasks to match and, also, the average request rate is proportional to the number of pilots but the frequency of the `TaskList` messages is not. The DHT broadcast structure allows the master to communicate with a high number of nodes without any increase in the number of messages it has to deal with. Moreover, the expected delay for the replies is of logarithmic growth (due to the properties of the Kademlia broadcast).

The same scaling pattern can be appreciated in Figure 10.4, which compares workflow turnaround times for the different cases[1]. The plot shows softer differences than those of inter-tasks delay for small testbed sizes because task execution times are basically identical for all setups. However, as the number of pilots and tasks increases, the results for pre-DHT and centralized configurations worsen as quickly as before. This is due to the inter-tasks delay telling only half of the story of the TQ congestion. For the complete picture, we must refer to Figure 10.5, which summarizes the most relevant data for an individual serial-chain workflow run of 10,000 tasks and 1,500 pilots, using the pre-DHT architecture.

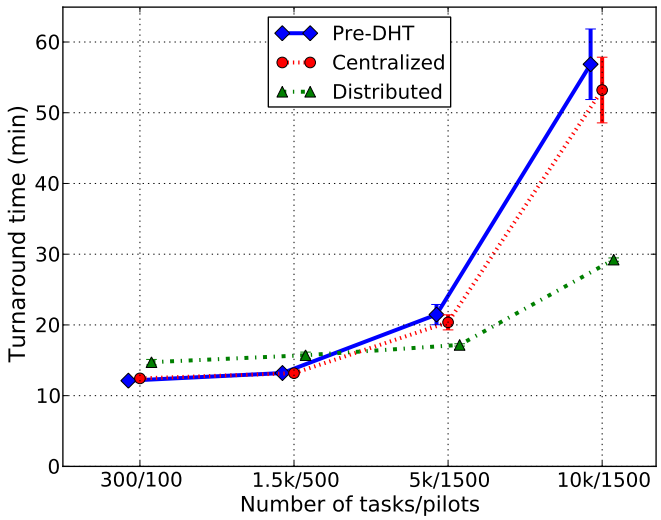


Figure 10.4: Turnaround workflow time per architecture and testbed size.

[1]Notice that in the last configuration shown in the plot, the number of tasks is doubled from the previous setting (5,000 to 10,000) but the number of pilots is kept constant. For this reason, the turnaround time increases significantly for all architectures.

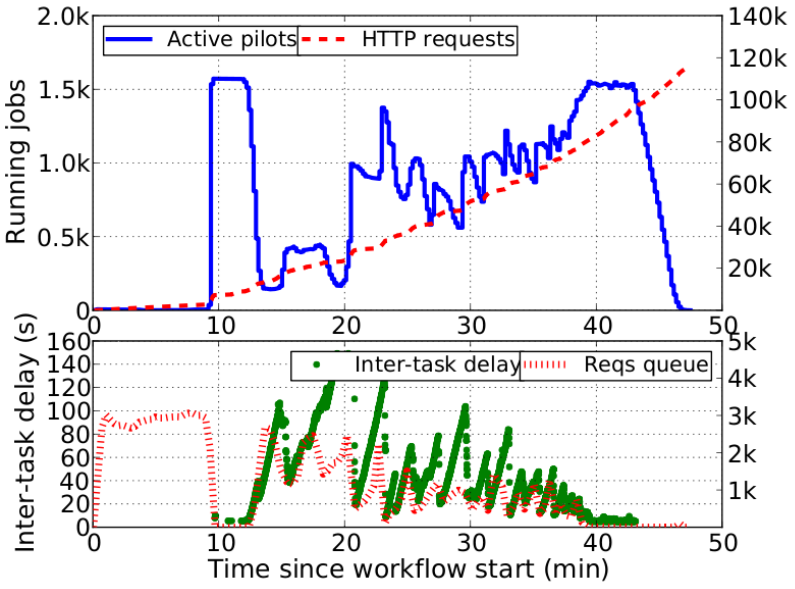


Figure 10.5: Pre-DHT architecture: slot occupancy and cumulative TQ requests (top), inter-tasks delay and requests queue (bottom).

The upper plot of Figure 10.5 shows the number of running pilot jobs versus time. When the workflow starts, all the pilots in the system contact the TQ to request their first task and download it at barely the same time. The TQ becomes overloaded and the time spent on each request rises. It takes the TQ almost 10 minutes to serve them all. Only then can pilots retrieve and start their tasks. This initial congestion is not reflected by the inter-tasks delay metric because only the interval *between* tasks is measured and these are the very first tasks on each pilot. The lower plot displays the correlation of the TQ activity with the inter-tasks delay and the length of the queue of pending requests. It also gives a feeling of the unpredictability of the scheduling overhead with this configuration.

For comparison, Figure 10.6 shows a run of exactly the same characteristics of the previous one but using the distributed architecture. This time, the number of running pilots and the inter-tasks delay follow a much more regular (and healthy) pattern and the queue of pending requests at the TQ is consistently kept very low during the lifetime of the workflow.

10.4.2. Cache Hit Ratio

Since the original motivation for the TQ architecture was to reduce the pressure put on SEs, it is important to evaluate how the new architecture

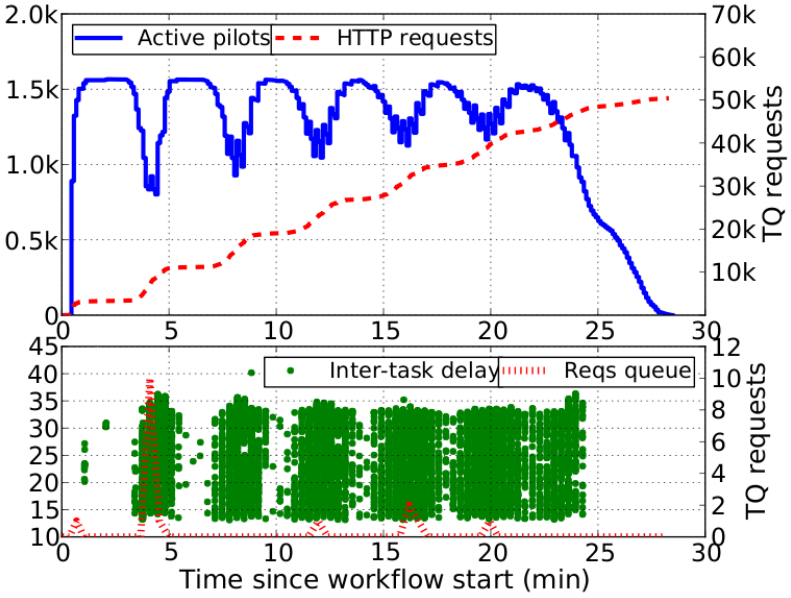


Figure 10.6: Distributed architecture: slot occupancy and cumulative TQ requests (top), inter-tasks delay and requests queue (bottom).

behaves with regard to the ratio of files read from the DHT/local cache in relation to the total number of reads.

Figure 10.7 displays the cache hit ratio for every architecture and testbed size. The configurations using the DHT have an almost perfect record of cache reads for every case because tasks can retrieve files from other nodes. With the pre-DHT architecture however, the hit ratio is considerably lower since tasks can only access files on its own disk. It is interesting to note that the hit ratio increases for larger testbeds. For brevity's sake, we will not get into details here, but we are aware of some race conditions that may cause a cache miss on the first second-step task that is run on a pilot. Since in the large testbeds the ratio of tasks to pilots is higher, these misses have a smaller relative impact.

Figure 10.8 shows the ratio of tasks reading from the local cache. The pre-DHT configuration behaves exactly as in Figure 10.7 since no DHT reads are possible in this case. This time, however, the centralized architecture obtains similar results. This is natural since both setups use the same matching algorithm. For the distributed case, though, the hit ratio is much better and it does not vary much with the testbed size. As discussed in Section 10.3.1, the master matches all known tasks and pilots at regular intervals and it can perform a better global matching than when serving requests one at a

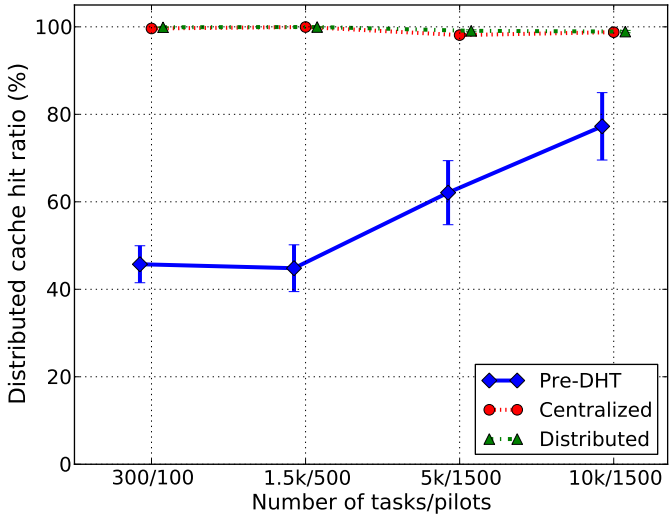


Figure 10.7: Distributed cache hit ratio per architecture and testbed size.

time. The race conditions mentioned earlier for the first set of second-step tasks are less significant with this algorithm and the results are similar for workflows with either low or high number of tasks.

Notice also that the dispersion of results for every case is relatively high. This is due to the lower outcome of the merging workflows in comparison to those obtained with serial chain and splitting. This is reflected by Figure 10.9, which shows local cache hit ratio results again but this time only for the distributed architecture and separating the different workflow types. The ratio for the merging workflow is significantly lower than the other two. The reason for this is that merge tasks must read two input files and, in general, these files will be located at different pilots. In cases like this, DHT reads are the only option to reduce the load on the SE.

10.4.3. Distributed Matching Ranking

As described in Section 10.3.1, the TQ distributed scheduling uses a non-exact matching algorithm, which provides a good but not optimal solution (in order to decrease its response time and make it usable for real-life work). Still, this procedure provides better results than the one used for centralized scheduling. In this section, we present some tests that assess and compare both algorithms.

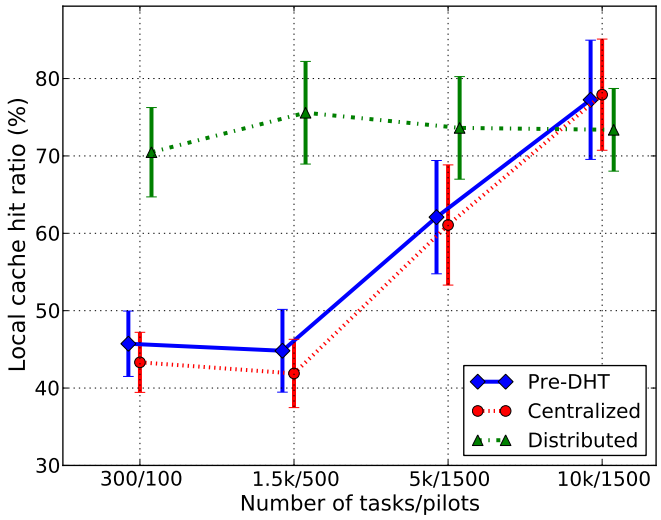


Figure 10.8: Local cache hit ratio per architecture and testbed size.

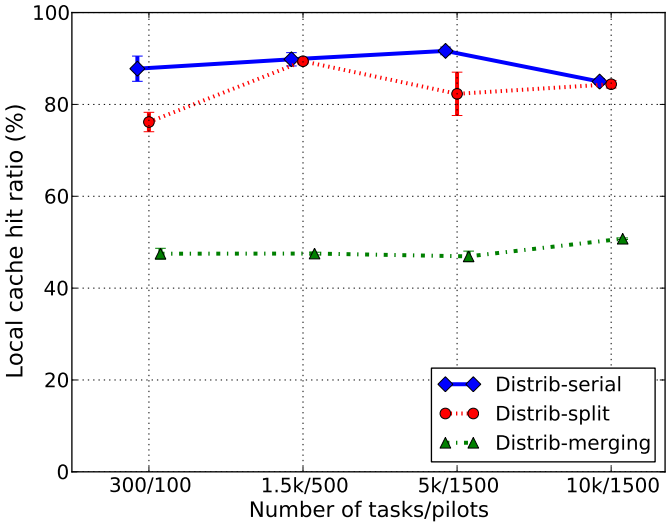


Figure 10.9: Local cache hit ratio per type of workflow and testbed size.

10.4.3.1. Tests Description

The TQ scheduling problem can be reduced to the maximization of a rank value resulting from the addition of the individual ranks assigned to the different pilot-task pairs (with the restrictions that one cannot assign a task twice or assign two tasks to a single pilot). For a given problem, we can express the different assignment combinations as a matrix of size NxM with one row per pilot (N), one column per task (M) and rank values as matrix elements. We can then apply our algorithms with such matrices as input and obtain a rank score (the higher the rank, the better the algorithm). Also, we can calculate the real maximum by applying a LP (*Linear Programming*) solver, in order to have an ideal rank to compare with. Notice that we cannot simply use this LP solver in real production since, for large matrices, this algorithm takes inadmissible long times.

In order to assess the quality of our algorithms, we will apply them to different test matrices of varying sizes. We will measure the rank obtained with each algorithm and how long it takes to achieve that result. We will use two types of matrices. First, matrices with random elements (from 0 to R_{max}; R_{max} being the maximum individual rank). In a second run, matrices with most elements having value of $\frac{R_{max}}{10}$ but a few ($\approx \frac{1}{N}$) having a value of R_{max}. The rationale is that most tasks can run on any pilot (low but non-zero rank) but one of them is better fitted for the task (R_{max}). Since this is not deterministic, for a few tasks, all pilots have a low rank; also for a few tasks, there may be more than one pilot with a high rank.

10.4.3.2. Results with Random Matrices

The results for the rank assessment with random matrices are shown in Figure 10.10. The delay incurred by each algorithm is displayed by Figure 10.11. We see that the ranks achieved by both the distributed and the centralized algorithms are very close to the absolute maximum of the LP solver. The delay however is vastly lower (about two orders of magnitude) for our algorithms. The distributed procedure gets results only marginally worse than the centralized one.

Notice also that in the real TQ deployment the central matching does not just resolve a matrix (as it is done in these tests), but it also needs to evaluate the requirement and rank expressions to build the matrix. This has been earlier shown to lead to congestion of the TQ and motivated us to develop the distributed algorithm in which those (possibly complex) logical evaluations are delegated to the worker pilots (the master needs only deal with the numerical matrix).

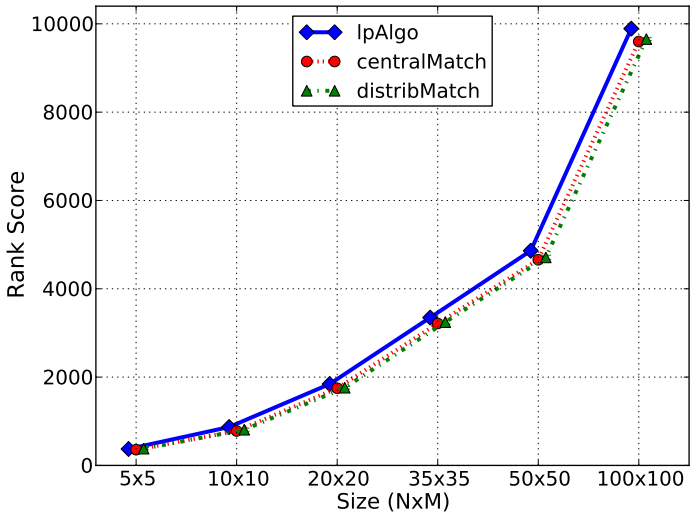


Figure 10.10: Rank score for different matrix sizes using random values.

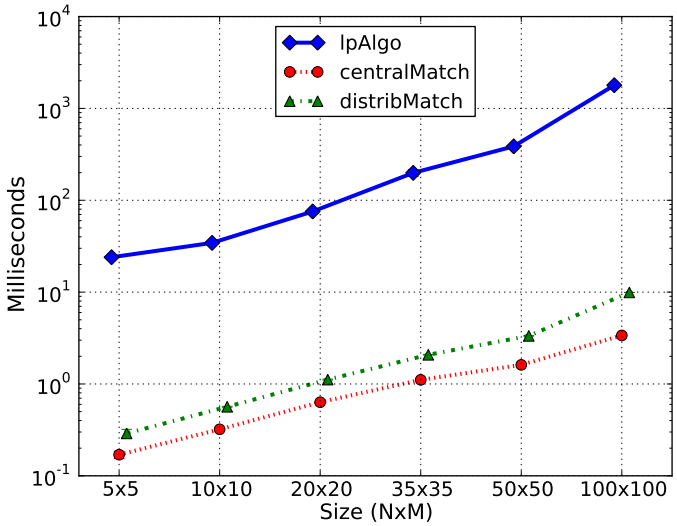


Figure 10.11: Algorithm delay for different matrix sizes using random values.

10.4.3.3. Results with Realistic Matrices

If we now repeat the tests with matrices containing more realistic values (as discussed above), we obtain the results shown by Figure 10.12 (ranks) and 10.13 (associated delays). In this case, we find a very similar distribution of delays (no significant difference between the distributed and centralized algorithms) but a higher difference in the rank results. With a less homogeneous input, choosing the maximum of one row (*centralized*) instead of the maximum of the whole matrix (*distributed*) may have more harmful effects if we lose a few high ranked pairs for the global sum. In a random matrix, however, the maximum values of the columns are not so different, in average, from the maximum values of the matrix.

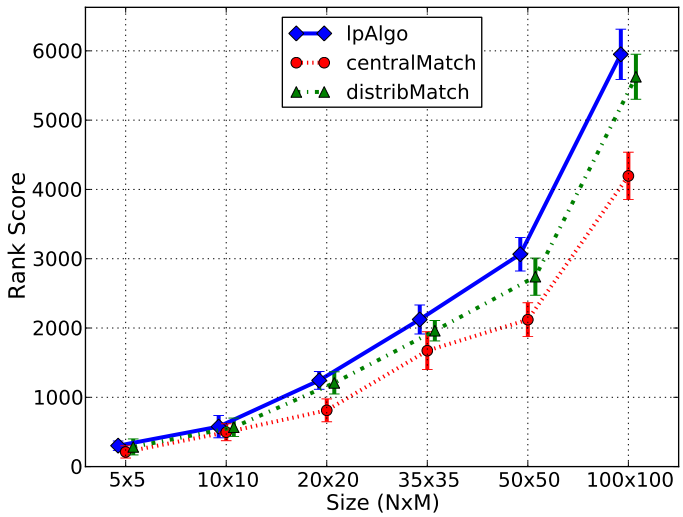


Figure 10.12: Rank score for different matrix sizes using realistic values.

In any case, what we can say is that the rank of the distributed algorithm is very close to the optimum value (LP solver), so it seems good enough for our purposes.

10.4.3.4. Real-Life Measurements

These tests were meant to compare rank values among the algorithms. As indicated, the delay values are only illustrative because we have verified that the LP algorithm is unusable in practice and because the central matching times cannot be reduced to this matrix resolution (it is better to compare distributed and central matching delays by running real workflows as done in previous sections). However, just to assess the applicability of the distributed algorithm to a real use case, we have run the test with a matrix of

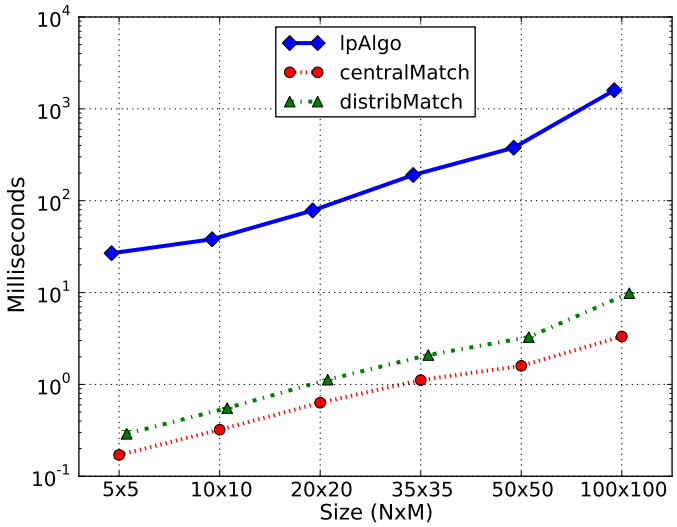


Figure 10.13: Algorithm delay for different matrix sizes using realistic values.

1,000x5,000 (i.e., 1,000 free pilots and 5,000 queued tasks) in a busy worker (running 8 simultaneous simulation jobs) of our infrastructure (8 Intel Xeon X5560 @ 2.80GHz and 24 GB of memory). The computation took around 9 seconds, which seems perfectly acceptable for such dimensions. We can compare this with the results shown in Section 3.3.2, where the matching of an equivalent matrix of 1,000x2,500 (half our problem) required 333 seconds to be performed [65].

10.4.4. Pilots Autonomy from Task Queue

Since one of the objectives of the new architecture was to make the system more robust against disruptions in the connectivity between the pilots and the TQ component, we made an additional modification in the design. Namely, pilots will now contact the master to report task completions and it will be the master which tells the TQ, at some later moment, in bulk. The advantage of this configuration is that, when a pilot finishes a task, it can immediately get new work from the master, even if the TQ is not available at the moment. The new architecture is reflected in the interaction diagram on Figure 10.14 (compare to the previous one in Figure 10.1).

In order to check how the system behaves against downtimes of the TQ, we have performed some tests in which the component is stopped for some time while a workflow is being executed on a set of pilots. For these tests, we have simulated downtimes of the TQ but not of the file servers from where the tasks sandboxes are downloaded or where the tasks completion reports are uploaded to. As discussed earlier, one can easily replicate these

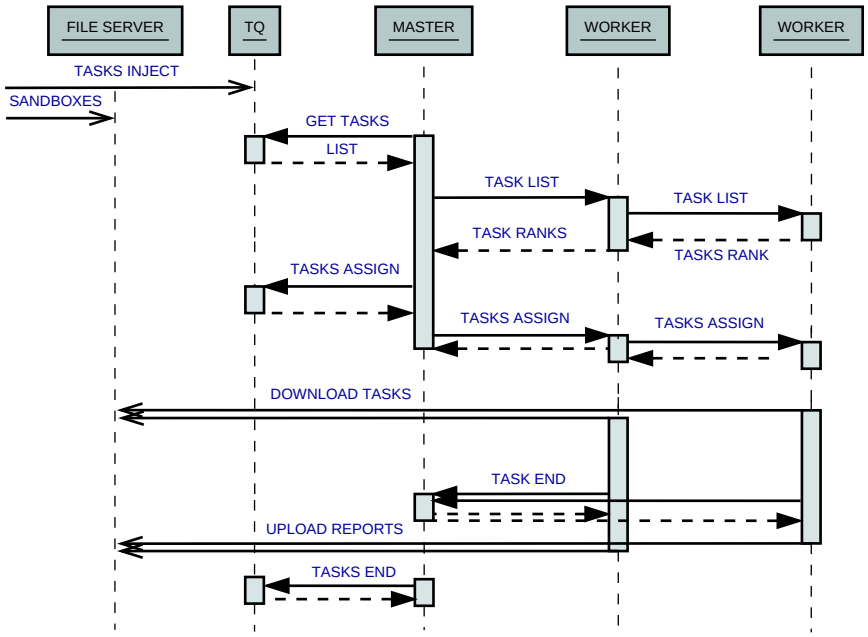


Figure 10.14: Enhanced distributed scheduling process.

file servers, since they entail no logic (this is only contained in the TQ) and HTTP caches per site (or at the master) could also be used. At this point, we have focused on decoupling the parts that require logical processing.

10.4.4.1. Tests Description

We have run a simple 2-step workflow of 320 tasks of 120 seconds each, on a site with 100 available job slots, and we have manually interrupted the communication with TQ in several different scenarios: the TQ is always available, the communication is interrupted 3 times for 45 seconds each, one single time for 4 minutes, twice for 4 minutes and, finally, a single cut of 10 minutes. We have run the workflow 3 times for each architecture (pre-DHT, central matching, distributed matching and the new one, with task ends reported by the master) and each scenario.

In order to better understand these processes, let us point at the detailed occupancy plots. Firstly, the one for the distributed scheduling is shown by Figure 10.15. In this plot, the communication cuts are shadowed in red. We can see that when the TQ is not reachable, the tasks can continue running (their completion is reported when the communication is recovered) but new ones cannot be started.

Now, for the new architecture with bulk task completion reporting, the situation is different. This is shown by Figure 10.16. This time, since the master requests more tasks than pilots are available in the system, these

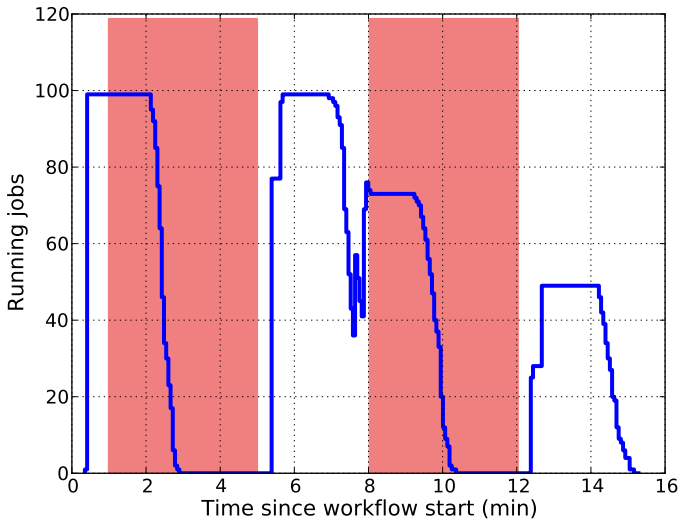


Figure 10.15: Slot occupancy on TQ disconnections with old architecture.

tasks can be started while waiting for the TQ to recover. In this case, one can also clearly see that the workflow completion time occurs at a time when it cannot be reported to the TQ and, thus, it will only learn that the workflow is done when it has come back (at the end of the red region).

10.4.4.2. Results

A word of caution: the tested workflow and breakdown scenarios are only some of an infinite number of possibilities. The results here presented are only illustrative. If the communication with the TQ were lost a bit earlier or later than occurred in our tests, the workflow completion times for one or other architecture might vary. The selected tests however succeed in showing a general trend in the behaviour of each architecture in case of problems.

Figure 10.17 compares the workflow completion times for the different architectures and scenarios. If no communication break takes place, the classical setups behave better than the distributed one because, for these very light loads, the overhead imposed by the bulk scheduling (at discrete steps) and the loss of a computing node (the master) weigh more than the gains they offer. Interestingly, the new system behaves like the previous version of the distributed matching (so there is no penalty for the bulk report).

When TQ disconnections occur, all setups suffer additional delays (but never a total breakdown of the system). The distributed architecture behaves only slightly better than pre-DHT and central matching (it is more

Figure 10.16: Slot occupancy on TQ disconnections with new architecture.

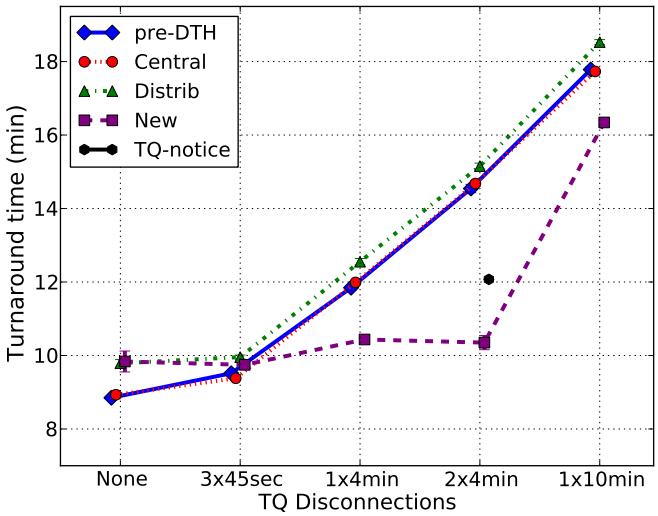


Figure 10.17: Workflow turnaround time for different configurations and TQ disconnection conditions.

resilient to short communication breaks but it cannot deal with longer cuts). The only one performing significantly better than the others is the new architecture, in which pilots can run tasks already known by the master with no need to talk to the TQ. Eventually, the master will run out of tasks and the sites will no longer be able to process further work (this is clearly seen in the last scenario, $1x10min$). However, the effect is delayed. Thus, the system is more robust than the previous versions.

In the scenario $2x4min$, we have depicted an additional plot point (labelled *TQ-notice*). This indicates the time when the TQ notices the workflow completion. This comes from the fact that the TQ cannot know that the workflow has ended until the site can communicate with it again. However, the workflow was completely finished earlier (*New*) and the pilots might have started new tasks if these had been already known by the master.

We would like to remark that, relative to the length of the tasks of this workflows, these are really long breakdowns. In particular, 10 minutes is more than the turnaround time of the whole workflow. If we scale task duration by a factor 100 (3.3 hours instead of 120 seconds), a 45 seconds cut would scale to 1.25 hours and the 10 minutes cut would represent one of 16.6 hours. It is therefore not surprising that the effective execution times are greatly affected by the downtimes. We can actually regard as a great success that the new architecture can potentially tolerate network disruptions of hours with no noticeable effect. Finally, let us point out that depending on the setup used with the pilots, such a long communication break could also cause the pilots to conclude that the TQ has definitively been lost and abort all operations. This would put an upper limit on the tolerable TQ disconnections. For these tests, we have used very high time limits so that pilots survive even the longer cuts because the results otherwise would have been trivial: pilots abortion and undefined workflow completion times.

10.5. Other Tests

Apart from the tests evaluating the general performance of the new TQ architecture and the accomplishment of the explicit targets of its design, some other experiments have been performed with the infrastructure. They are described in the following sections.

10.5.1. Task Length and Workflow Turnaround Time

In Chapter 5, we introduced a model for workflow execution in grid resources. The model was further discussed in Section 8.2.1, where we derived an expression to compute the optimal task length for workload division, given the inter-tasks delay (time between consecutive tasks) caused by match-making and task start-up.

In order to informally assess the validity of our model, we have studied one of the runs of the experiments discussed in Section 10.4 and see how its results fit our equations. We have selected one of the runs of the serial chain workflow, with 10,000 tasks and around 1,500 pilots, and the distributed architecture (i.e., like the one evaluated at Figure 10.6). Since each of the tasks has to perform useful work for approximately 211 seconds and, in this case, the exact number of available slots was 1,565, we can compute the constant introduced for Equation 5.14 as:

$$C = \frac{W}{\varphi K} = \frac{10000 \cdot 211}{1565} = 1348.24 \tag{10.1}$$

On the run, we have measured an average initial delay of 29.5 seconds and an average inter-task delay of 25 seconds. This means that our theoretical makespan, given by Equation 5.14 results in:

$$L = 29.5 + \frac{211}{2} + 1348.24 \cdot (1 + \frac{25}{211}) = 1643 \tag{10.2}$$

In reality, the workflow took around 1,700 seconds. The discrepancy is caused by the fact that the last steps are not filling the 1,565 slots completely (because 20,000 is not divisible by 1,565) and so, in reality, $avg(T_i^e) > \frac{T^t}{2}$. In any case, we are not very far from the theoretical result.

We can also calculate the optimum task size for this case using Equation 8.3 and find the corresponding total workflow duration in that case:

$$T^t = \sqrt{2 \cdot 25 \cdot 1348.24} = 259.6 \tag{10.3}$$

$$L = 29.5 + \frac{259.6}{2} + 1348.24 \cdot (1 + \frac{25}{259.6}) = 1637.4 \tag{10.4}$$

Hence, the real turnaround time is close to the optimum value. If, as an example, we had chosen a task duration of 50 seconds, the workflow would have lasted considerably longer:

$$L = 29.5 + \frac{50}{2} + 1348.24 \cdot (1 + \frac{25}{50}) = 2076.9 \tag{10.5}$$

10.5.2. Data Access Patterns

Congestion problems on SE access are often caused by heterogeneous data access patterns, especially the so-called *hot spots*, in which many tasks consume the same (small) set of input files. A test case was designed to study how our system behaves in such situations.

10.5.2.1. Tests Description

In this test, a 2-step workflow is run on 50 pilots. In the first step, 20 tasks produce 20 files (one each). On a second step, 140 tasks access 2 of those 20 files. This means that each file is accessed 14 times. We run with three different architectures to see their different behaviours: pre-DHT, centralized and distributed.

In the worst case scenario, each of the original 20 pilots would need to serve the same files to 14 pilots. However, we will see that the TQ pilots do better than that. Firstly, because some of the tasks run on pilots with their required input files in their local cache. Secondly, because as soon as a pilot has retrieved a file from a peer, it can serve it as well, so the number of sources for the file increases (and requests are split among all the pilots serving a file at a given moment).

10.5.2.2. Results

The plot at Figure 10.18 shows the distributions of reads (from SE, from local cache and from other pilots using DHT) for each one of the configurations and for the two workflow steps. What we see is that, in the pre-DHT case, the SE is accessed for around 127 files (an average of 6 times for each of the files produced at the first step). With the sharing of files among pilots (DHT), these accesses are avoided. So, first of all, the possibility of creating hot spots on the SE itself is greatly reduced (which was the main target of the whole system). The figure also confirms that the local hit ratio is higher for the distributed cache than for the centralized one.

To also understand what happens with the files served by DHT, we must turn to the distribution of served files per pilot. This is shown by Figure 10.19, for centralized matching, and Figure 10.20, for the distributed case, which shows a very similar plot, with the difference that the number of files that need to be transferred via DHT is lower.

We see that the number of pilots having served any of the files is higher than the number of original holders (20); i.e., 34 in the case of distributed architecture and up to 46 in the centralized case. Besides, the average number of files transferred by a single pilot (line with triangles as markers) is considerably lower than if only those 20 sources had been used (line with circles as markers). As a reference, the average transfer value for the case where all files were transferred using the DHT (what we earlier referred to as *worst case*) is also plotted (line with diamonds as markers).

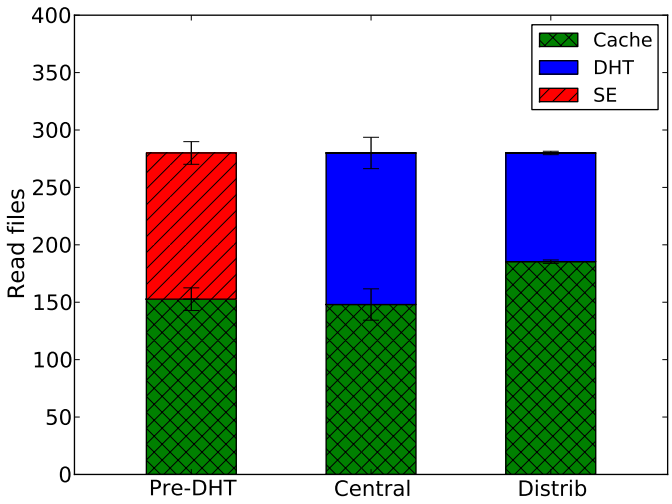


Figure 10.18: Read distribution for different cache configurations.

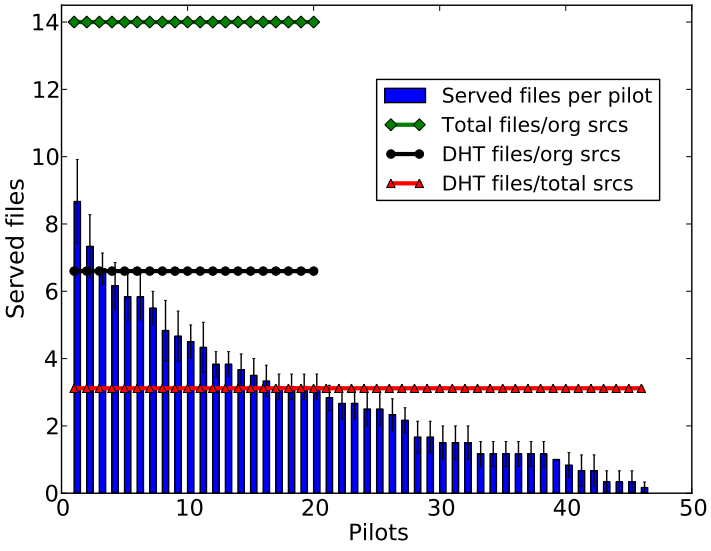


Figure 10.19: Distribution of files served per pilot with centralized matching.

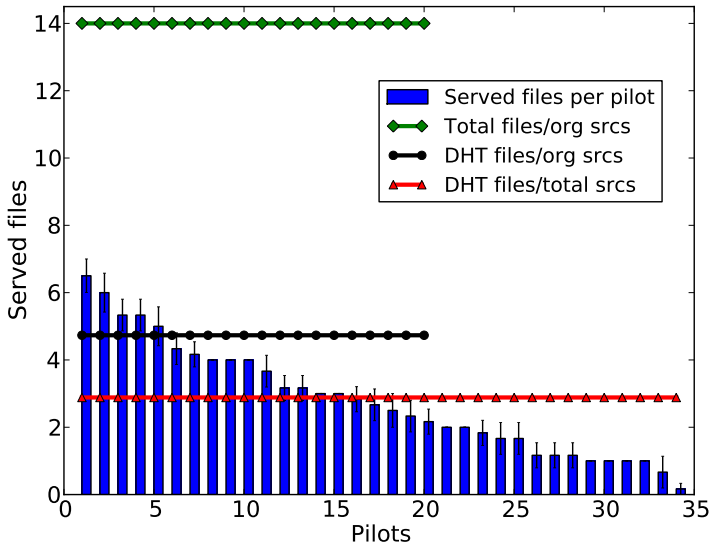


Figure 10.20: Distribution of files served per pilot with distributed matching.

10.5.3. Operation in a SE-less Resource Center

10.5.3.1. Sites without Storage Element

As discussed in Section 5.1, scientific grid workflows are subject to different data access paradigms. It cannot be ruled out that pilots are run on computational resources with no local access to massive storage (e.g., WLCG's Tier-3 centres or cloud machines). The tests discussed in this section assess the behaviour of the TQ architecture in such circumstances.

Our tests simulate a workflow execution in a site without SE, where data is accessed through the WAN on a remote SE. The data processing tasks do not just copy the remote file, but open it and read it remotely. Depending on the format of the file and the configuration, they may need several round-trip-times to access different parts of the file. For this case, we will not try to model the data access but just assign some inefficiency to the processing time, so that the task is slower. For merge tasks, we will just use a WAN transfer rate since they do not process the data, they just get the files to merge them. CMS has verified that opening and reading a file for merging (as opposite to copying it) is very inefficient due to the high number of required round-trip accesses.

For the simulation to be realistic, we would like to replicate what CMS jobs experience on their remote interactions. However, these numbers are not easy to determine. They depend on many factors, which vary wildly. Indeed, the performance depends on the network connections between local

and remote sites, the congestion of WAN and LAN links at a given time and the stress conditions of the remote SE. We do have some numbers from some CMS internal reports. Sadly these are CMS private communications, so they are only available for members of the collaboration. For example, one document[2] indicates that for some workflows, remote analysis incurs a 10% penalty. However, in another case[3], for pile-up events, the efficiency (relation between processing time and total job time) falls to around 30% compared to the LAN case.

10.5.3.2. Tests Description

We have run a 2-step workflow on a site of reduced capacity (25 available slots only). In the first one (processing), 120 tasks read 3 files each from a remote SE. There are only 70 different files, so each file is read around 5 times, for a total of 360 read operations. Each processing task produces 1 small output file. In the second step (merge), 4 tasks consume all the output files (30 files each) to create 4 big files and store them at the SE. We have run this for an architecture with no cache at all, for the pre-DHT configuration and for the distributed one.

In order to cover all the spectrum of possible performances, we will simulate that the efficiency of tasks reading remote data is one of 30%, 60% or 90% while tasks reading local data (on pilot's local cache) have a 100% efficiency. Meanwhile, for the complete file copies, we will set a transfer rate of 5 MB/s (a typical CMS SE-to-SE rate, see Section 6.1). In our case, copies are actually performed by the merge tasks, running on the WN, so the average transfer rate should be significantly lower than between SEs (specially because they need to copy many small files). This means that our results (on how advantageous the pilots cache is) are actually conservative.

10.5.3.3. Results

Figure 10.21 shows the distribution of reads (from SE, from local cache and from other pilots using DHT) for each one of the configurations and for the two steps. When no cache is in place, all file reads imply accessing the remote SE. When a cache is present (either with or without DHT), more than half of the files are read from the local disk of the pilots, for the processing step. If the DHT is present, then some additional files are read from other pilots (only the initial pilots need to retrieve the files from the remote SE, i.e., around 70 files). For the merging step, only a few files can be read from the local cache because the merge task needs to retrieve all the files produced at all the pilots. It is in this case where the DHT is more useful, since it avoids interacting with the remote SE completely.

[2] `https://indico.cern.ch/event/343183/contribution/0/material/slides/0.pdf`
[3] `https://indico.cern.ch/event/341563/contribution/4/material/slides/0.pdf`

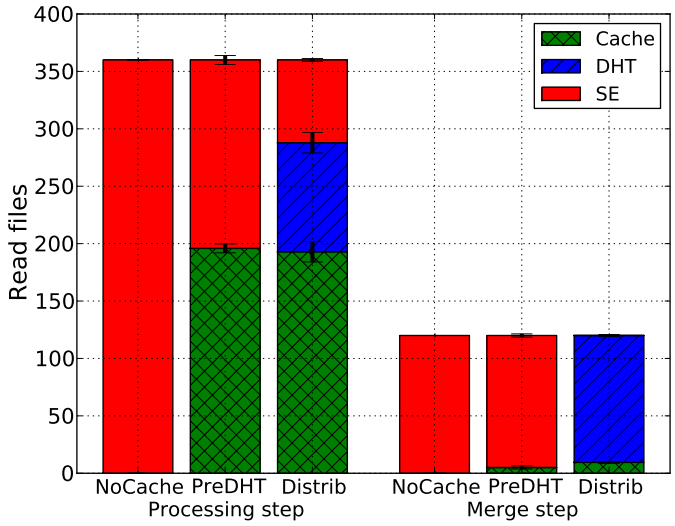


Figure 10.21: Distribution of read operations for different cache configurations.

Figure 10.22 shows the effect of the remote interactions in the total workflow completion times, for each setup and tested efficiency values. For comparison there is an additional result labelled with a 100% efficiency, which corresponds to an ideal workflow run on a local SE, without congestion and optimal network connectivity. The figure also displays the duration of a merge task as a small light grey bar superposed to the bottom part of the larger bars. Let us recall that the disparities are produced by the different efficiencies in the processing tasks (which is reflected in the final turnaround time) and also, though less noticeably, by the lower transfer rates when downloading remote files (merge tasks). The latter is reflected by the shorter merge tasks when DHT is used.

For a 90% efficiency, all the configurations behave in a very similar way and are not too far from the value obtained for the ideal local workflow. However, for lower efficiency values, the differences grow. This was expected and depends on the type of workflow being run but also on the particular characteristics of the sites involved in the operation. What we can say is that the use of a cache, especially the distributed one, protects us against low processing efficiencies, reducing their effect in the workflow completion times. In addition, we must not forget that WAN access may cause additional problems in practice (broken connections, really slow network links). This means that the improvement produced by the cache should be, in general, higher than the one shown in the plots.

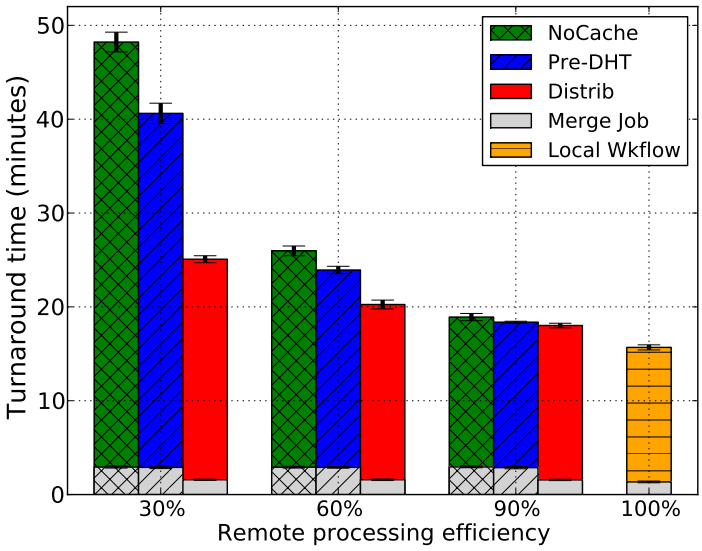


Figure 10.22: Workflow turnaround time vs remote processing inefficiency for different cache configurations.

WRAP-UP: *We have presented a new architecture that, by creating a DHT system among pilot nodes, is able to offer an effective data cache and a distributed scheduling mechanism, which is far more scalable than its centralized alternative. The new system is also more robust: there is less interaction between the sites and the central queue, which has become simpler and less loaded.*

We have shown that this model produces limited and predictable scheduling overheads, even at large scales, and that this leads to shorter workflow execution times. We have also verified that, by performing bulk task matching at regular time intervals, the global assignment rank can be increased; in particular, the hit ratio of our data cache rises, even for small workloads. Furthermore, DHT-based file sharing services turn the local cache into a distributed one, with an almost perfect hit score. In addition, several different tests have shown that the new architecture is better prepared for difficult scenarios such as suffering disruptions in the connectivity to the central TQ service or the absence of a local storage system on the site where a workflow is being run.

Part IV

Conclusions

Chapter 11

Conclusions

11.1. Conclusions

This thesis has studied the scheduling and execution of large data-intensive workloads on distributed computing resources. We have analyzed the main issues involved in the management of large amounts of data and the techniques required to efficiently access and process it. The central work of this thesis is the design, implementation and evaluation of a scalable architecture for the completion of workflows with heavy data requirements, the *Task Queue*. This new system introduces the concept of micro-scheduling, according to which, individual nodes—rather than sites—are selected as destinations for computational tasks. The architecture also improves data access by using a distributed data cache and considering data location for scheduling at both the grid and the cluster level. As a second major contribution, we have performed an in-depth analysis of the broadcast operation in the Kademlia DHT, since this functionality was required for our distributed task matching algorithm.

Concerning scheduling, we have shown that the location of data is a major factor to take into account, not only to optimise job efficiency, but also to avoid excessive data replication and the problems that this entails (results published in [112]). However, coordinating data placement and workload management is a highly complex task and current approaches in WLCG tend to separate both concerns even if the scheduler is able to suggest the replication of certain data based on the requirements of already queued tasks.

Our newly developed architecture builds on existing late-binding overlays, which have been highly successful in WLCG. Among other things, these solve several issues shown by traditional scheduling: excessive dependency on precise and up-to-date resource information and inability to enforce VO priority policies. In order to understand early- and late-binding scheduling better, we have enhanced an existing performance model for both approaches

and we have reached an analytical expression that provides the time required to complete a workload when discrete tasks are run. This expression allows us to calculate the theoretically optimal task length value and lets us understand the importance of reducing the scheduling overhead to avoid limiting the scalability of the whole system (these results have been submitted to a journal for publication and are pending acceptance).

A key piece of the Task Queue architecture is the Kademlia network shared by the pilots at a grid site. We have used this DHT to build a distributed data cache among the pilots (they can share files without the need of a central catalog) and to implement a cooperative task matching and assignment procedure. For this algorithm, we had to implement the broadcast primitive on top of the DHT routing facilities. In fact, we have performed a deep study of the problem of broadcasting in Kademlia. We have analytically and experimentally proven that full coverage with the minimum number of messages is achievable with the only use of the DHT contact tables. We have shown that this is possible with our bucket-based algorithm and existing partition-based protocols, though not for an arbitrary splitting degree. In addition, we have provided a thorough study of the performance of each protocol in the presence of churn or failure conditions and suggested different techniques (notably, redundancy and/or resubmissions) to improve the results in those cases (published in [101] and [113]).

Many different tests have been carried out to assess the performance of the Task Queue. We have shown that the use of a data cache produces a reduction in the number of accesses to the massive storage systems and that this translates into shorter workflow makespans, especially when working under very demanding conditions—e.g., with a very stressed or absent local storage service (published in [114]). It has also been demonstrated that the introduction of the DHT-based shared data cache is very useful to increase the hit ratio for certain data dependency patterns. Finally, and most importantly, the distributed task assignment algorithm produces improved global assignment rank values while successfully increasing the scalability of the system—due to limited and predictable scheduling overheads—as well as the autonomy of the pilots from the Task Queue (results included in [115] and in the article pending publication).

11.2. Outlook and Future Work

As we have seen throughout this work, efficiently scheduling and executing large workloads with substantial data requirements is a challenging task. However, the increasing abundance of data at our disposal suggests that not only are data-intensive applications common today but they will be even more frequent in the future. Therefore, the optimization of that kind of workflows will become even more appreciated.

While we are confident that we have offered some contribution to the field, this is an open problem and it will probably always be, given that the related technologies are constantly evolving. Therefore, we plan to continue working on this area and, in particular, on the topics dealt with by this thesis.

For instance, we aim to study the possibility to integrate external data location information into our task matching procedure. The most prominent use case would be the sites whose SE is a distributed file system on top of worker node disks (e.g., Hadoop). The SE files could be used for micro-scheduling just like cached files currently are.

Another idea we would like to explore is the construction of a grid-wide DHT network with a generalized data sharing protocol, where choosing one origin or another was just a question of preference. I.e., select first a pilot at the local node; otherwise, a pilot at the local site; if not possible, fall back to the local SE; and, finally, resort to a pilot or SE in another site. This would constitute a more elegant and homogeneous model than the one we have today. In principle, though, the cooperative task scheduling procedure would still lie confined to the boundaries of a site.

In more general terms, we aspire to deepen our understanding of how micro-scheduling and inter-pilot communication can be applied to the improvement of the execution of massive workloads in large, complex, distributed computing infrastructures.

Part V

Appendices

Appendix A

Implementation and Architecture Details

This appendix gives additional information on the details of the internal architecture and implementation of some systems discussed throughout the document.

Although the most relevant aspects of the Task Queue architecture were discussed in Chapter 7 and 10, many details of its implementation, internal architecture and the integration with existing job submission systems were left out for the sake of concision. The first sections in this appendix complete that information. Section A.1 gives details on the complete TQ architecture, Section A.2 deals with the implementation of the TQ component and Section A.3 discusses the algorithm and thresholds used by the Pilot Monitor component to ask for submission of new pilot jobs to grid sites.

As described at length in Chapter 10, the pilot agents are key for the newly proposed distributed architecture for data caching and micro-scheduling. Section A.4 studies their internal architecture and implementation.

Finally, Section A.5 and A.6 provide additional information on the testbeds used for the evaluation of the Kademlia broadcast algorithms and for the execution of non-CMS workflows, respectively. The first one was discussed in Section 9.5, while the latter was utilized for the tests shown in Chapters 7 and 10.

A.1. Complete Task Queue Architecture

The diagram in Figure A.1 shows the integration of the TQ system and the CMS scheduling system. The components labelled as part of the *classic PA* were those used by CMS to submit jobs directly to the grid resources (including the Tier-0). In order to link these with the new components (Task Queue, Pilot Monitor, Pilot Manager) the appropriate API (*Application Pro-*

gramming Interface) plugins are used.

In the figure, we can trace a CMS workflow from its injection into a PA instance. The PA makes use of a specific submitter plugin to enqueue jobs into the TQ in a controlled way. The PA also keeps track of the life cycle of the tasks and retrieves resulting logs and reports using TQ-specific plugins. From the point of view of the existing code, the TQ is seen as another grid site. Once in the TQ, the tasks are retrieved and executed by pilot jobs submitted by the Pilot Manager and monitored by the Pilot Monitor.

We are not really interested in the details of the ProdAgent (more so, since this system was phased out in favour of the new WMAgent—and its *WMCore* framework—and glideinWMS), but we want to stress that the introduction of the TQ did not require major modifications in this component. The TQ is just another resource to which jobs are submitted. It is interesting to note, however, that existing data dependencies between CMS jobs are translated into task requirements by the TQ plugin upon submission. Once in the TQ, the tasks and their requirements are managed independently of the client that submitted them.

It is clear, however, that in order to fully take advantage of the possibilities offered by micro-scheduling, the PA (or the relevant workflow manager or submission agent) must be made aware of the details of the TQ operation and must be coded explicitly to use the TQ interface appropriately. Let us not forget that, while the TQ enables us with powerful means to match and rank tasks and pilots, the TQ itself is completely ignorant about the semantics of the task requirements. It is the duty of the submission agent (who knows the work to be done) to set those properly.

The Pilot Manager submits pilots to the grid when this is requested by the Pilot Monitor. The latter queries the TQ about queued tasks and releases new pilots as a result. To do this, it will take into account how many pilots have already been sent to each site (active or still idle) and what are the thresholds for those sites. It uses information from the TQ and from the standard PA's JobTracking component, which tracks the pilots. A running pilot registers itself with the TQ, asks for a task, downloads its description and input sandbox, and runs it. When this is finished, it informs about its exit status and uploads the task output and task report. It also sends a heartbeat message every once in a while, reports unexpected errors and informs about its own termination.

The TQ is able to keep track of tasks and pilots, provides appropriate tasks for a pilot that requests so (matching the task requirements) and sets the correct status when the task is complete. It also logs important events in each pilot's lifetime and archive pilots that announce their termination or that have not sent a heartbeat for too long. The TQ also offers an API for tasks and pilots status retrieval.

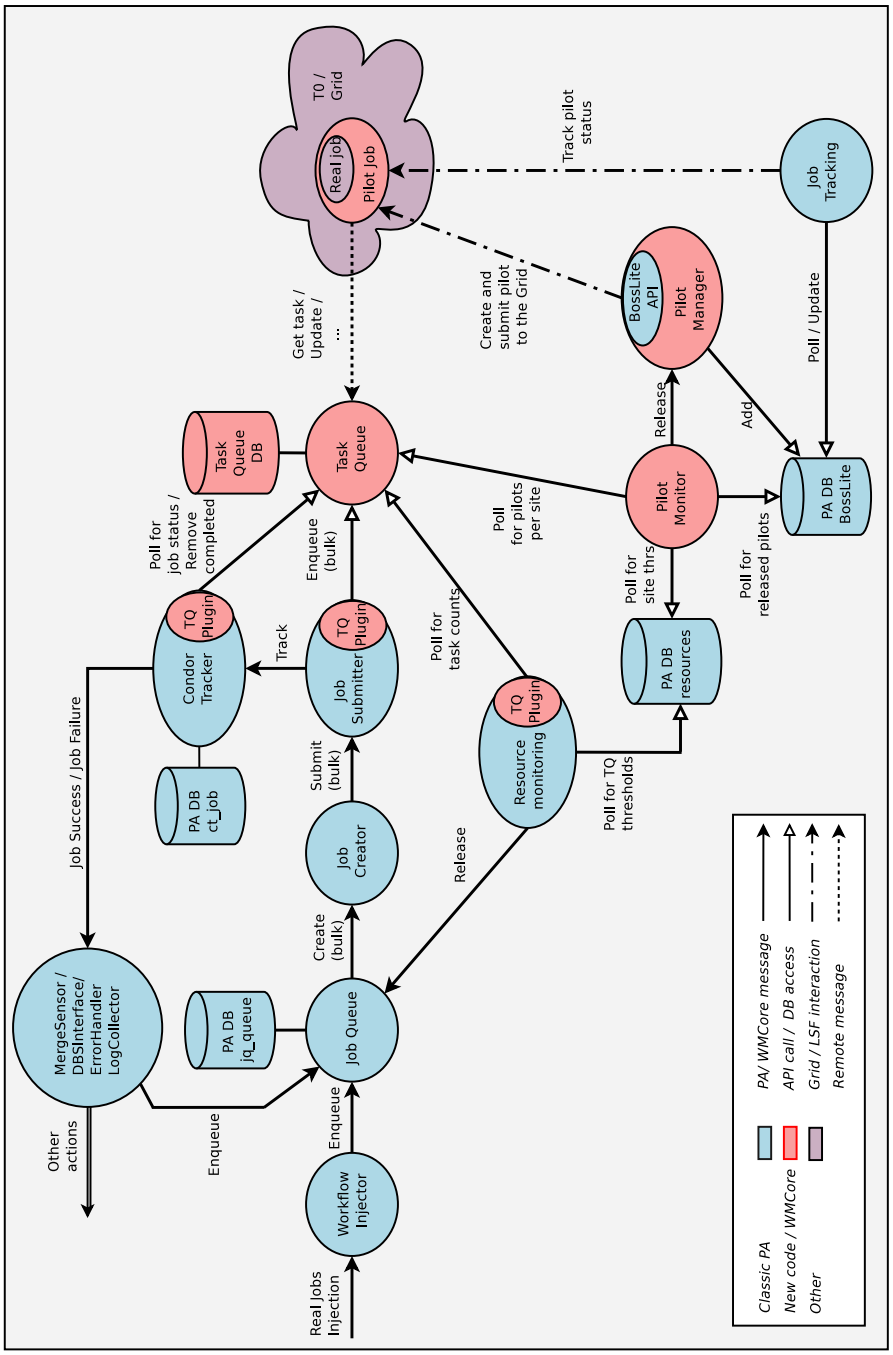


Fig. A.1: Integration of the Task Queue with the existing CMS submission system.

A.2. Task Queue Internals

Figure A.2 shows the main components of the more recent version of the TQ and its interactions with external agents, such as any WMCore component, the pilot jobs (via a REST interface) and any client using the public API, for purposes such as task enqueuing, monitoring or debugging. The TQ is started as a daemon using the WMCore command line client.

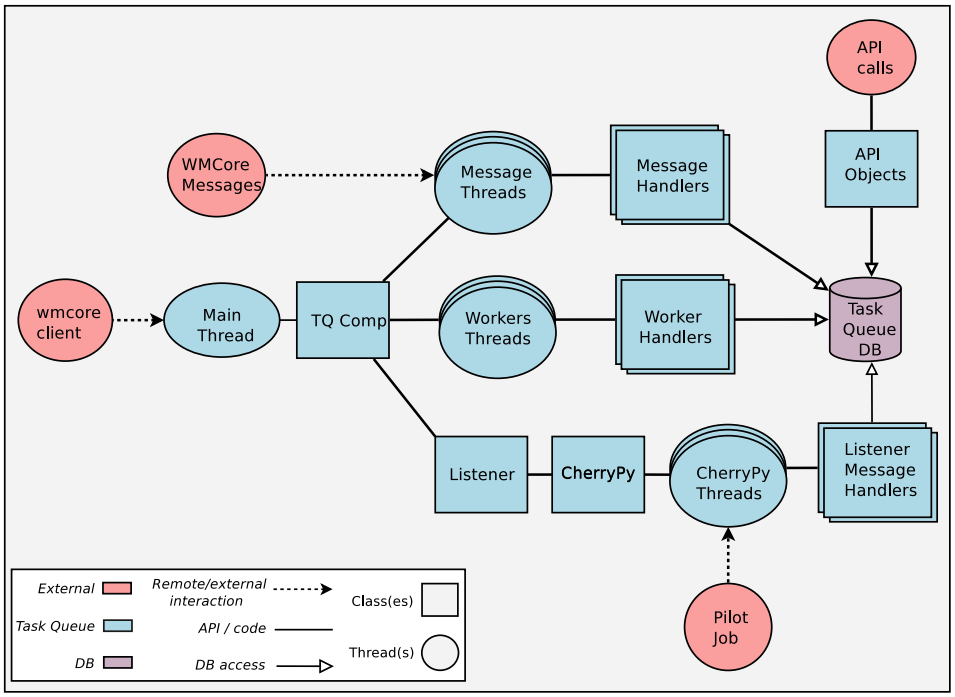


Fig. A.2: Diagram of classes and threads of the Task Queue.

Probably, the most striking feature of the diagram is the central role that the DB (*Database*) has in the architecture. In fact, all state information of the TQ service is persisted on the DB. This has several advantages. Firstly, this makes it easy for the several threads in the system to exchange information (thus, cooperate) in a safe way. An adequate use of transactions and locks avoids any race conditions. Secondly, if for some reason (e.g., maintenance) the service must be interrupted, the operations can continue almost seamlessly upon restart, because all state information can be recovered from the DB. Moreover, high availability and load balancing could be achieved by sharing the DB among several TQ servers. Finally, providing an API for external queries (e.g., for monitoring) or archiving historical information can be conveniently accomplished by coding the appropriate SQL (*Structured Query Language*) queries.

There are several parallel lines of execution within the TQ. Firstly, the

main thread is started when the TQ boots up and it is in charge of parsing the configuration information, setting up the necessary data structures for the other components (e.g., the map between message types and associated handlers) and starting the other threads. The *worker threads* take care of routine tasks, such as periodically verifying the heartbeat messages of the pilots in the system and removing them if they are too old. The *message threads* run the handlers associated to the WMCore messages[1]. A server running the CherryPy web framework [116] is responsible for the REST communication with the pilots. This server is composed of several threads that listen on the defined port and serve requests by running the handler that is appropriate for the received message type. This will act on the DB and send a reply to the remote pilot, as appropriate.

Finally, apart from the different thread pools running on the TQ, several API objects may be used by external clients to run a series of predefined queries on the TQ DB. Read-only and read-write interface objects have been defined separately so that, depending on the settings, clients may be authorized to just perform monitoring operations or, otherwise, modify the DB upon task insertion, removal or archival.

A.3. Pilot Release Algorithm and Thresholds

A.3.1. Pilot Release Algorithm

How many pilots are sent to each grid site at a given time depends on a number of factors. The Pilot Monitor component periodically considers all the available information and calculates the number of pilots that should be released at that moment. The Pilot Monitor owns the information about how many pilot jobs have been submitted to each site and how many of those have finished already. The rest are either running or queuing at the site. To complete this information, the Pilot Monitor queries the TQ about the tasks in the queue and the registered pilots.

In order to perform its duty, the Pilot Monitor needs to be aware of the destination of the queued tasks. Since this is an essential information, the TQ manages it as a special requirement (stored in a dedicated DB field) and exposes it through an API, used by the Pilot Monitor. This API provides the information of the number of tasks requiring each site and those that can run on *any* site. For example, the API could offer the following information:

```
[
  {"sites": ["site1", "site2"],  "tasks": 100 },
  {"sites": ["site1"],           "tasks": 50  },
  {"sites":  null,               "tasks": 250 }
]
```

[1] At the moment, this capability is not really used, but the machinery is ready.

The result (in JSON format) indicates that 100 tasks can run at *site*1 or *site*2, 50 tasks require *site*1 and 250 tasks can go to any site. With this information, the Pilot Monitor is ready to run his algorithm, once for each known site. The following listing summarizes its main steps in pseudo-code:

```
1.   Recall thresholds and previously submitted pilots for site
2.   Set:  available_slots := max_pilots - submitted_pilots

3.   If  available_slots <= 0:
4.       Then:  Do not continue (do not submit more pilots)

5.   Query TQ about tasks that can run on this site
6.   For each group of enqueued tasks:

7.       If  enqueued_tasks < inactive_pilots:
8.           Then:  Mark inactive pilots as active, mark tasks as covered
9.           Else:
10.              If  available_slots > number_of_tasks:
11.                  Then: send more pilots, mark tasks as covered

12.  If  idle_pilots < min_idle_pilots:
13.      Then: send more pilots

14.  If  submitted_pilots < min_submitted_pilots:
15.      Then: send more pilots
```

The algorithm shows a few per-site thresholds that are taken into account when calculating the number of pilots to submit. We discuss the pilot management thresholds in the next section.

A.3.2. Site Thresholds

Mimicking the way that PA throttled the submission of jobs to the sites (to avoid overloading them), the Pilot Monitor uses some per-site thresholds when requesting the submission of new pilots. At the moment, these are defined manually, but more dynamic ways of handling this are possible, based on the grid information system or on proper analysis of the number of idle or queued pilots at each site.

The initially defined thresholds were the following:

- `minimum/maximum_submission`: Minimum/maximum number of pilots to be submitted at once to the site.
- `max_pilots`: Maximum number of pilots simultaneously running or queuing at the site.
- `min_submitted_pilots`: If there are fewer submitted pilots, submit some more (up to the number of available resources for the site).

- `min_idle_pilots`: If there are fewer idle pilots, submit some more (up to the number of available resources for the site).

It may be worth indicating that the `min_submitted/idle_pilots` thresholds are used to submit pilots to sites even when no task is waiting to run there at a given moment. A legitimate reason for this is to have those pilots ready for new tasks arriving to the site, so that there is no initial delay in running them.

A.4. Pilots Internals

Figure A.3 shows a simplified diagram of the internal architecture of a pilot job. Everything is set off by an *invoking agent*, which is normally a script arrived as a grid job to a site's worker node. The pilot main thread initializes a file server (to share cache files with other peers) and the DHT subsystem, which fulfills the duties imposed by the membership to the Kademlia network.

The basic model of operation is based on alarms, timers and messages. When the pilot's main thread completes an action, it sets a timer for its next expected activity and goes to sleep until it is awaken by an alarm. An alarm may be caused by a timer previously set by the thread itself or may be the result of a DHT message having arrived from the DHT subsystem. In the latter case, the handler associated to the *type* of the received message is loaded and executed.

In stable conditions, the main pilot thread is looping in either the master or the worker main routines. In the first case, a new thread is launched every time a task is run and the main thread basically waits for it to finish (though it still reacts to alarms). In the second one, the main thread interacts with the TQ and the other pilots (making use of the DHT capabilities) and orchestrates the distributed task matching procedure.

The cache manager component takes care of the files in the peer's cache. For simplicity, not all possible access relations have been drawn in the diagram, but it is clear that both the file request handlers and the running tasks may access the files on the cache. Also, the main thread is able to contact other peers to fetch required input files and, subsequently, ask the cache manager to include them in the local cache.

As for the DHT subsystem, the core functionality is operated by the handlers invoked by the *Listener*, which runs an *asyncore* loop. Asyncore is a python module for asynchronous socket handling and it is used here to deal with simultaneous communications with multiple DHT peers and also with the local client (in this case, the pilot's main thread) [117]. When a new message is received, the appropriate handler (external for DHT messages, internal for local commands) is run. These handlers are able to set timers on

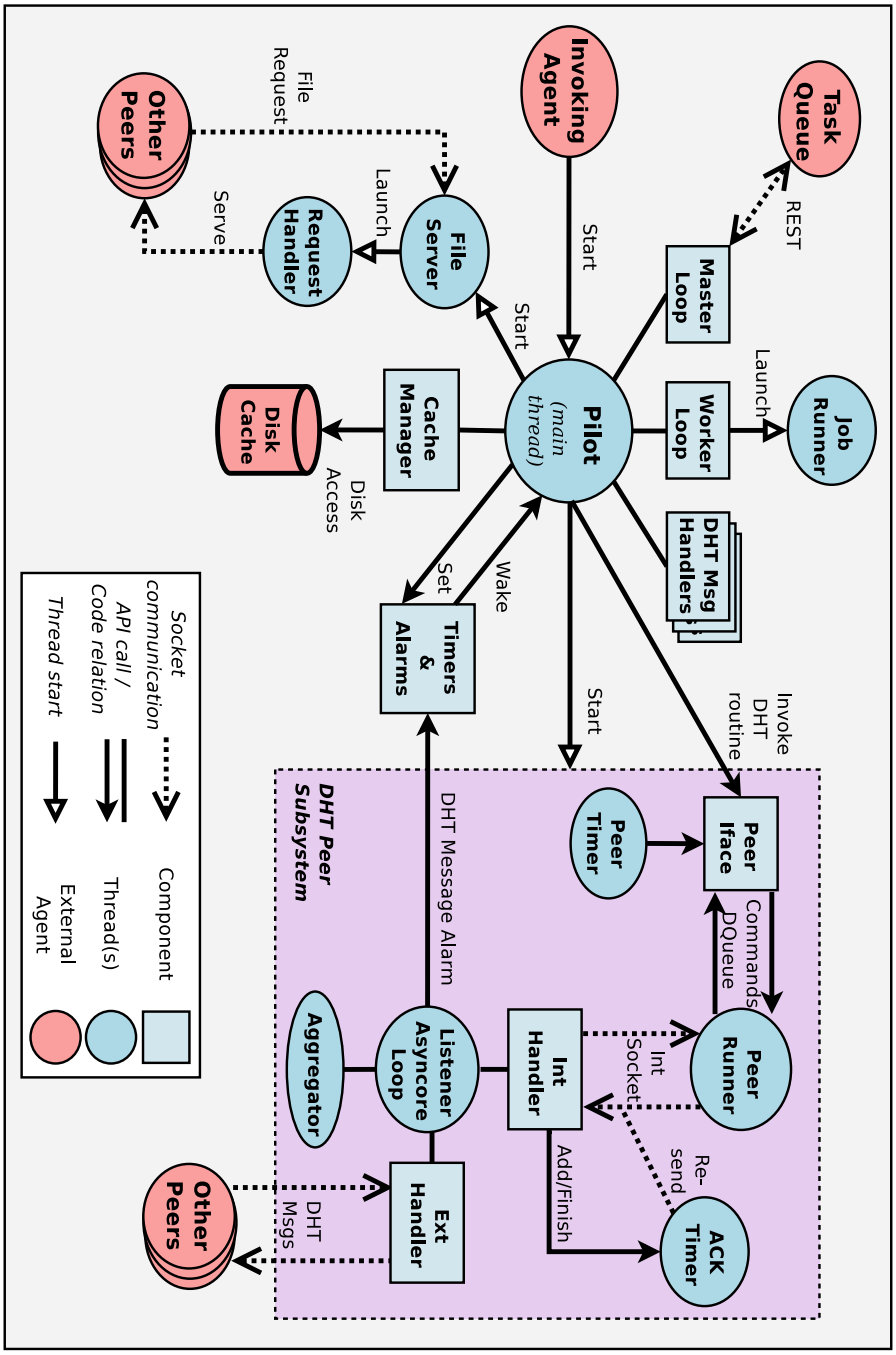


Fig. A.3: Internal architecture of the pilot agents.

the pilot's alarm table in order to communicate with it or to open sockets to remote peers when outgoing messages need to be sent.

The interface of the DHT peer exposes high-level methods, such as *join network*, *locate close peers*, *store value*, *retrieve value* or *send broadcast message*. These methods may be translated in several low-level operations (such as contacting more than one external peers). Every command received through the interface is put on a thread-safe duplex queue of tasks. We call this queue *duplex* because it supports the returning of results for completed operations: the client requests an operation and, if desired, may call a blocking operation to wait for a returned value. In this way, the peer interface may be used by several independent threads, which is actually the case here since both the pilot's main thread and the DHT's *Peer Timer* use it.

The other DHT-related threads are basically timers. The *Peer Timer* is in charge of Kademlia's periodic tasks, such as refreshing the routing buckets or republishing stored key-value pairs. The *ACK Timer*, in turn, keeps a table of the DHT messages that were sent and for which an ACK (*Acknowledgment*) message is due. If a timeout occurs for a message, this timer asks for its resubmission. The *Peer Runner* thread just executes the commands requested in the previously described duplex queue. It uses an internal Unix socket to send the necessary requests to the Listener thread.

Finally, the *Aggregator* keeps track of aggregated replies for broadcasts. This component maintains a table of messages for which reply values should be aggregated (e.g., a list of tasks per pilot for all the nodes in the network). Each entry in the table includes information such as the function to be used for the aggregation, the address of the peer node that sent the message to us, the number of replies that we should receive before sending the result back and a timeout value after which we should reply even if not all replies were received (this depends on the node's position in the broadcast tree). When a DHT message is received, the associated handler to be executed gets a reference to the Aggregator object so it can add or remove entries in the table or update existing ones. When due, the Aggregator thread will directly send aggregated results to the appropriate DHT peer.

A.5. DHT Testbed Internals

Figure A.4 shows a diagram of the deployment and duties of the different elements involved in the DHT broadcasting tests. The tests were run on a parallel facility at the CIEMAT institute, called *Euler*. Euler currently runs a batch system so that independent batch jobs can be run on the offered slots. For our tests, we launch 50 independent jobs on 50 slots. Each one of these spawns 20 independent DHT peer threads, which join the Kademlia network initiated by previously started peers. The result is a Kademlia DHT with 1000 members.

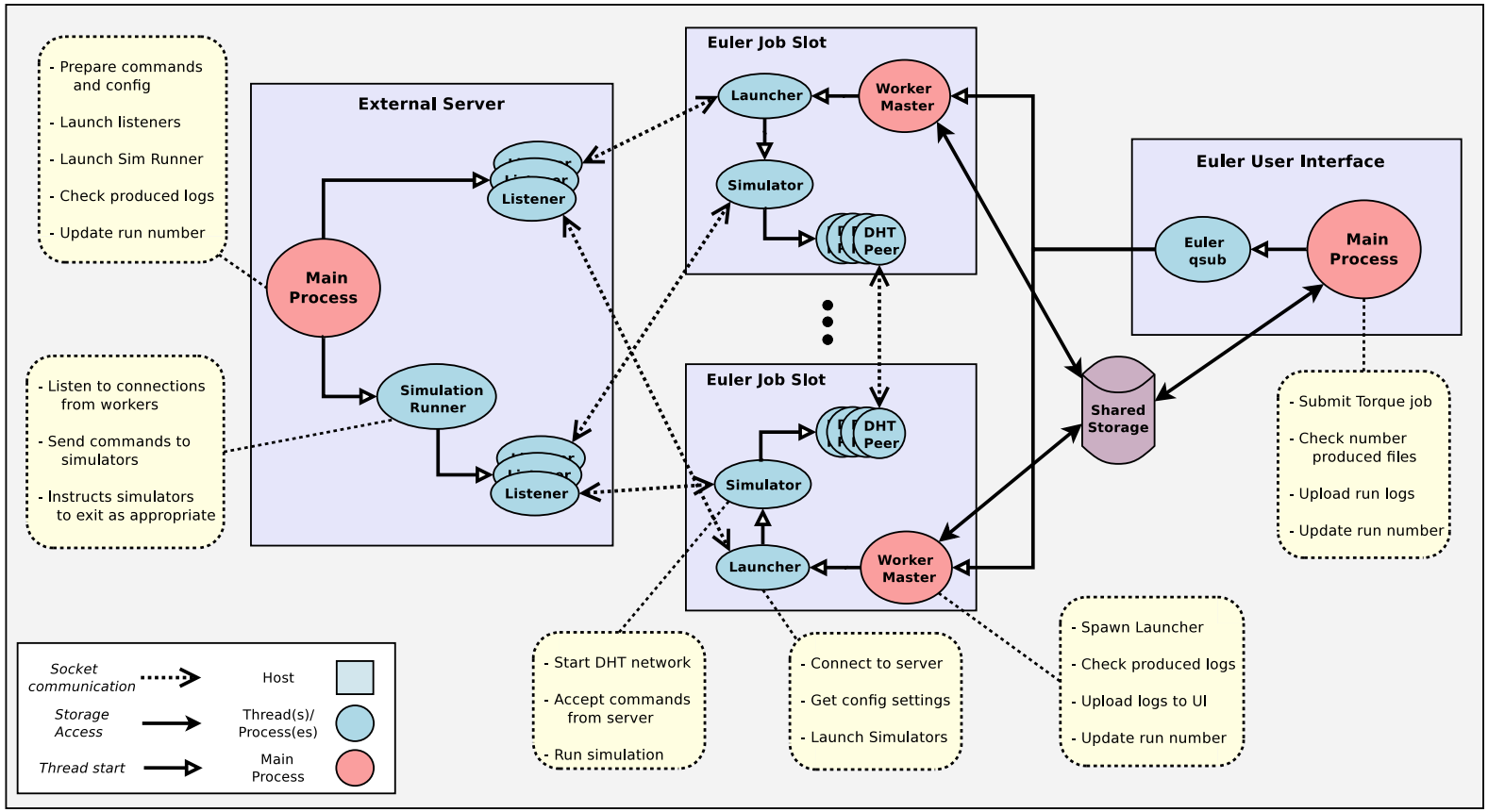


Fig. A.4: Architecture of the testbed used in the evaluation of the DHT broadcast algorithms.

In the figure, we can also appreciate some processes running on two hosts external to the Euler nodes. The first one of these is a user interface set up for job submission to Euler. A single process runs here, with the duty of, firstly, launching the jobs (with a `qsub`), watching their produced log files, detecting when a given test run has completed and, before a new run is started, packing all the resulting logs for off-line parsing and analysis. This is possible because Euler provides shared storage between the workers and the UI (*User Interface*), so that the jobs can write their logs there and the process at the UI consume them.

There is another external entity running a server whose purpose is to feed the DHT simulators with consistent configuration information and the commands to be run. For each set of tests, a configuration file with the necessary settings is prepared for the server. The settings include the initial number of DHT peers to be run, the rates of peer joining and leaving the network, the frequency of broadcasts and the total length of the test. The server parses this configuration file and translates it into commands that the remote simulators can run (e.g., *add another peer* or *tell peer X to initiate a broadcast*).

Like in the case of the UI, the external server also verifies the completion of test runs in order to proceed with the subsequent ones when due. In this way, long series of DHT broadcast tests can be run unattendedly.

A.6. Non-CMS Testbed Internals

Figure A.5 shows a diagram of the components of the testbed used to run non-CMS workflows with the TQ. The elements in blue are the ones composing the TQ architecture and are independent of the type of workflow being run. The pink components are those developed to manage and execute non-CMS workflows.

The initial script `runMany.sh` parses a single configuration file indicating a series of tests and their characteristics (number of steps, tasks, dependencies, etc.). For each test run, it invokes the *workflow manager*, which runs a complete workflow and, then, another script `processRun.sh`, which parses the produced log files. When this is done, it goes on to the next run and the process is repeated.

The workflow manager creates the tasks of the first step, based on the parameters given by the initial script and submits them to the TQ. From this moment, it polls the TQ to keep track of the evolution of all the tasks. As these finish, the workflow manager creates new tasks, depending on them. Since the non-CMS tasks are instrumented to provide any monitoring information that is interesting for our analysis, the `processRun.sh` script can easily parse them and produce the desired statistics and plots. This is done on the fly for each workflow run.

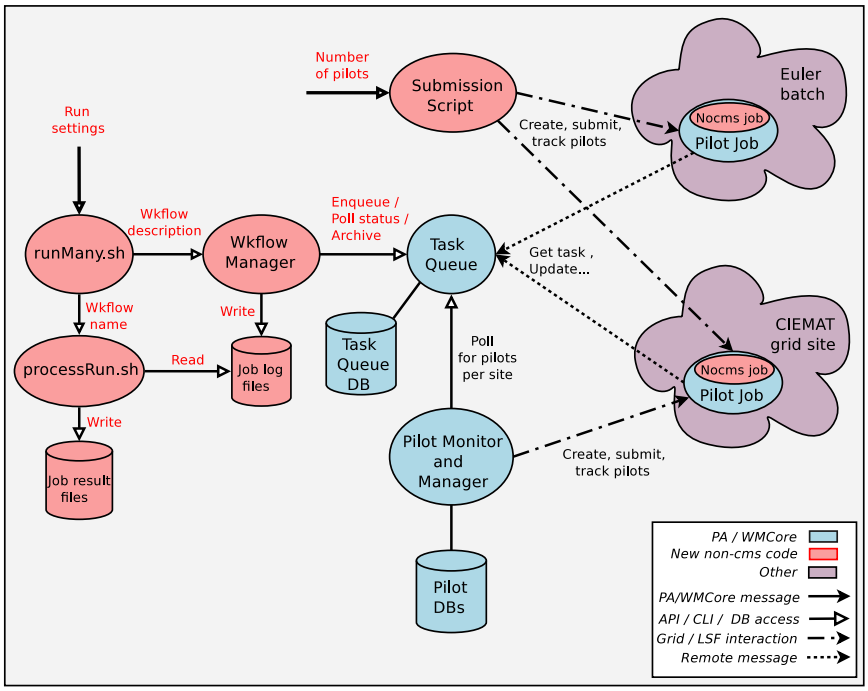


Fig. A.5: Components of the non-CMS Task Queue testbed at CIEMAT.

Regarding the management of pilot jobs, two different submission methods were used. While for tests in Chapter 7 we had the Pilot Monitor decide how many pilots to send based on queued tasks, for the tests in Chapter 10 a different approach was followed. Since we were focused on the performance of the TQ component and the interactions among the pilots, we had no particular interest in verifying the Pilot Monitor further. Moreover, we required additional resources, therefore we looked at methods to incorporate the slots in the CIEMAT's Euler facility (with no grid interfaces) to our tests. We decided to develop some simple scripts that were able to schedule and start the desired number of pilots, either on Euler slots or on the grid site worker nodes and operate them manually. The pilots were started prior to the tasks being queued and were able to run several different workflows sequentially. This kept scale testing manageable.

Resumen de la tesis

Esta tesis estudia la planificación y ejecución de trabajos computacionales intensivos en datos a gran escala. En ella se analizan los principales problemas relativos a la gestión de grandes volúmenes de información, así como las técnicas necesarias para procesarlos. Su principal contribución es el diseño, implementación y evaluación de una arquitectura escalable para completar flujos de trabajo con grandes requisitos de datos. Este nuevo sistema, al que hemos llamado *Task Queue* (*TQ*), introduce el concepto de *microplanificación*, según la cual, se seleccionan nodos individuales como destino de las tareas computacionales, en lugar de centros de recursos (*sites*), como se hace tradicionalmente en las infraestructuras *grid*. Nuestra arquitectura también mejora la eficiencia de acceso a datos, al construir una *cache* distribuida y utilizar la ubicación de los ficheros, tanto en el ámbito del *grid* como dentro de cada *cluster*, como parte del proceso de planificación. Una segunda aportación a destacar en el trabajo es el análisis en profundidad de la operación de *broadcast* en *Kademlia*, un importante sistema de tabla *hash* distribuida (*Distributed Hash Table*, *DHT*), motivado por la necesidad de utilizar esta funcionalidad en nuestro algoritmo cooperativo de asignación de tareas.

Vivimos en la era de los datos. Tanto en el ámbito empresarial como en el social y el científico se generan ingentes cantidades de información, cuya gestión requiere enormes capacidades de procesado, transferencia y almacenamiento. En el ejemplo paradigmático de la computación científica, un amplio rango de infraestructuras distribuidas son utilizadas hoy en día para hacer frente a las necesidades de los experimentos. Entre ellos, las infraestructuras *grid* constituyen probablemente el ejemplo más representativo de un entorno complejo de computación distribuida. El *grid* aglutina numerosos centros de recursos en múltiples dominios administrativos, enlazados por complejas redes de comunicación, que no siempre pueden garantizar bajas latencias. Las aplicaciones en el *grid* se enfrentan a interfaces y entornos heterogéneos, incertidumbre sobre los recursos disponibles y la inevitabilidad de una cierta tasa de error por configuraciones deficientes,

problemas de *hardware* y actividades de mantenimiento.

Nuestro trabajo arranca en el entorno del *Worldwide LHC Computing Grid* (*WLCG*), actualmente el mayor *grid* computacional en el planeta, y en concreto en el experimento *Compact Muon Solenoid* (*CMS*), como respuesta a ciertos problemas de congestión en el acceso a los sistemas de almacenamiento que ralentizaban la realización de sus trabajos computacionales. A partir de ahí, afrontamos el problema general de la planificación y ejecución de cargas de trabajo intensivas en datos. Consideramos que los problemas y soluciones encontrados pueden ser interesantes no solo para el entorno actual de CMS o incluso para el contexto de los *grid*s de uso científico, sino para ámbitos más generales.

En lo referente a la planificación de tareas, nuestro trabajo muestra que la ubicación de los ficheros a procesar debe ser un factor fundamental a tener en cuenta, no solo para optimizar la eficiencia del acceso a los datos, sino también para evitar una excesiva replicación de los mismos y los problemas que esto conlleva: uso innecesario de los recursos de red y del espacio de disco, aumento de la presión sobre los sistemas de almacenamiento (*Storage Element*, *SE*s). Esto fue confirmado en un trabajo temprano consistente en el desarrollo y prueba de una versión mejorada del metaplanificador *GridWay*, que incorporaba información sobre los datos requeridos por las tareas en el proceso de su planificación.

Sin embargo, es interesante resaltar que la coordinación del emplazamiento de datos y la gestión de las tareas a ejecutar no es una actividad sencilla. Las predicciones sobre la duración de las tareas individuales son habitualmente poco fiables o inexistentes y las políticas de emplazamiento de ficheros pueden llegar a ser muy complejas. Las aproximaciones actuales en WLCG tienden a separar ambas responsabilidades: los ficheros en general se replican en función de su popularidad pasada (sin evaluar las necesidades presentes), aunque en ciertos casos sí se permite que el planificador realice sugerencias de replicación de datos en base a los requisitos presentados por las tareas encoladas para ejecución.

La arquitectura de la TQ extiende los modelos de asignación tardía, basados en agentes llamados *pilots*, cuya misión es recuperar carga de trabajo real de una cola de tareas ubicada en un servicio central de una organización virtual (*Virtual Organization*, *VO*). Los sistemas de *pilots* han tenido un gran éxito en WLCG por varias razones. En primer lugar, eliminan la incertidumbre sobre el estado de los recursos disponibles que presentaba el envío tradicional (directo) de trabajos, puesto que solo solicitan una tarea cuando ya están en ejecución. En segundo término, la cola central de ta-

reas dota de flexibilidad a las VOs para actualizar sus políticas de prioridad internas de manera sencilla. Además, los *pilots* permiten asegurar que el entorno de un nodo es el adecuado para las necesidades de ejecución antes de comenzar esta y ofrecen un interfaz homogéneo a las tareas de la VO.

Para comparar mejor los modelos de asignación temprana y tardía, hemos perfeccionado un modelo existente sobre el rendimiento de ambos, alcanzando una expresión analítica que indica el tiempo requerido para ejecutar un conjunto de tareas de duración discreta, en función del tiempo de planificación. Esta expresión nos permite calcular el valor teórico óptimo de la duración de las tareas y nos muestra la importancia de reducir el sobrecoste temporal que el proceso de asignación conlleva, si queremos evitar limitar la escalabilidad de todo el sistema.

El objetivo inicial de la nueva arquitectura era aprovechar los sistemas de *pilots* para crear una *cache* que ayudara a mejorar la eficiencia del procesamiento de datos y redujera la presión que el acceso simultáneo de un gran número de tareas causa en los sistemas de almacenamiento. Para conseguir que la *cache* sea efectiva, es necesario que las tareas a realizar se asignen a los nodos de ejecución apropiados: aquellos que albergan los ficheros de entrada requeridos. La microplanificación resulta pues necesaria para esta meta. Pero el potencial de esta técnica de asignación a nodos individuales va más allá de este uso particular. Gracias a ella podrían seleccionarse *pilots* en función de la memoria, el disco, el número de procesadores o instrumentación asociada. Igualmente, se podría, por ejemplo, equilibrar el número de tareas limitadas por entrada/salida y por procesamiento en un mismo nodo.

El problema con esta idea es que las colas centrales de tareas pueden convertirse en cuellos de botella para el sistema y limitar su escalabilidad. La cantidad de peticiones que el servidor TQ debe responder aumenta con el número de *pilots* en ejecución. Además, un mayor número de tareas en cola y una mayor complejidad de los criterios de selección suponen más candidatos que inspeccionar para cada petición y más tiempo invertido en la evaluación de cada uno, respectivamente. Los sistemas de *pilots* existentes atajan el problema agrupando bien tareas con requisitos comunes bien *pilots* (por ejemplo, en *sites*) o usando una asignación previa de tareas a *pilots*, antes de que lleguen a ejecutarse. En estos casos se está efectivamente restringiendo (o abandonando) la microplanificación o la asignación tardía.

La versión final de la arquitectura TQ utiliza una red Kademlia formada por los *pilots* de un *site grid* para crear una *cache* distribuida de datos (gracias a la cual los *pilots* pueden descubrir y compartir ficheros) y para implementar un procedimiento cooperativo de asignación de tareas. De este modo, el servidor TQ central no necesita mantener un catálogo con la ubicación de todos los ficheros en la *cache* y tampoco debe encargarse de las

costosas evaluaciones de las tareas para emparejarlas con los *pilots*. En lugar de ello, un *pilot* maestro solicita de la cola central todas las tareas a ejecutar en su *site* y después difunde la lista de tareas entre todos los *pilots* para que cada uno asigne un rango de preferencia a cada una de ellas. A continuación estas puntuaciones son devueltas al nodo maestro que las compara y realiza el emparejamiento definitivo.

El algoritmo de asignación de tareas descrito requiere el envío de la lista de tareas a evaluar a todos los *pilots* en la red Kademlia. Es decir, necesita del envío de mensajes de *broadcast*. Sin embargo, esta funcionalidad no está presente en el protocolo de Kademlia original así que ha sido necesario añadirla, aprovechando las capacidades de enrutamiento del DHT. De hecho, esta extensión de su protocolo nos ha llevado a un estudio en profundidad del problema de *broadcast* en Kademlia. Nuestro trabajo muestra tanto analítica como experimentalmente que se puede conseguir una cobertura perfecta con un número mínimo de mensajes usando simplemente las tablas de contactos que el DHT ofrece. Esto se puede hacer con nuestra propuesta de que un nodo reenvíe los mensajes recibidos a un contacto de cada *bucket* de Kademlia, o bien usando alguno de los algoritmos existentes de división del espacio de identificadores para el reenvío. Este último caso, sin embargo, solo es válido, como demostramos, si se usa un valor de división compatible con los mencionados *buckets*.

Nuestro estudio sobre el *broadcast* en Kademlia se completa con un análisis de los problemas derivados de posibles errores de transmisión y de una alta rotación de nodos de la red. Ofrecemos una fórmula para la cobertura del envío en función de la probabilidad de error en cada mensaje y señalamos que los árboles de difusión con menos nodos en las primeras etapas obtienen mejores resultados. Además, proponemos y evaluamos varias técnicas de mejora cuando se dan pérdidas de mensajes. Los principales ejemplos son el uso de mensajes ACK directos y retransmisiones, y el envío simultáneo de varios mensajes redundantes (aprovechando las características particulares de Kademlia). Finalmente, mostramos que es necesario escoger la política de reenvío adecuada para mensajes duplicados, puesto que, de lo contrario, las técnicas de mejora pueden tornarse ineficaces o incluso perjudiciales.

Se han realizado numerosas pruebas para evaluar la funcionalidad y el rendimiento de la *Task Queue*. Se ha mostrado que el uso de una *cache* de datos produce una reducción en el número de accesos a los SEs de los *sites* y que esto supone un menor tiempo de ejecución global, especialmente si se trabaja en condiciones exigentes (con un SE muy cargado o en centros que no disponen de servicio de almacenamiento local). También se ha demostrado

que la introducción del sistema distribuido mejora significativamente el ratio de aciertos en la *cache* para ciertos patrones de dependencias entre tareas (por ejemplo, cuando una tarea necesita leer datos producidos por varias antecesoras, ejecutadas en nodos diferentes).

Como resultado fundamental, hemos comprobado que el procedimiento de asignación de tareas distribuido consigue aumentar la escalabilidad del sistema. Esto se debe a la enorme reducción en el número de interacciones con el servidor TQ y al hecho de que las tareas más complejas quedan a cargo de cada comunidad de *pilots*. De esta manera, nuestra arquitectura puede crecer hasta un elevado números de *pilots* y tareas sin renunciar a la microplanificación ni a la asignación tardía. Además, el nuevo sistema de asignación periódica, en grupo, consigue un rango global más alto, puesto que puede optimizar las asignaciones a varios *pilots* a la vez, en lugar de considerarlas secuencialmente. Por último, el nuevo sistema mejora la autonomía de los *pilots* y reduce la dependencias con el servidor TQ. Todo esto se ha podido comprobar con diversos experimentos en los que se medía la respuesta del sistema a gran escala, se cuantificaba el rango global de las asignaciones y se comprobaba el efecto que un corte en la conectividad entre los *pilots* y la cola central tenía sobre el rendimiento global.

A lo largo de toda la tesis, comprobamos que la planificación y ejecución eficiente de grandes cargas de trabajo con enormes requisitos de datos supone un desafío. Sin embargo, la creciente abundancia de información a nuestra disposición sugiere que las aplicaciones intensivas en datos, ya frecuentes hoy en día, lo serán más todavía en el futuro, por lo que su optimización será cada vez más necesaria. Estamos convencidos de haber realizado algunas aportaciones al estudio de este problema. Sin embargo, este continúa siendo un campo abierto de trabajo y probablemente lo seguirá siendo indefinidamente, puesto que las diversas tecnologías implicadas evolucionan constantemente. Por este motivo, planeamos seguir trabajando en esta área y, en particular, en algunos de los aspectos tratados en esta tesis.

Nos gustaría estudiar la posibilidad de integrar información externa sobre ubicación de ficheros en nuestro procedimiento de asignación de tareas. La principal aplicación de esta funcionalidad se daría en aquellos *sites* que utilizan un sistema de ficheros distribuido sobre nodos de ejecución como SE (por ejemplo *Hadoop*). De esta manera, sus ficheros estarían disponibles para nuestra microplanificación de la misma forma que los ficheros de la *cache*.

Otra idea que desearíamos explorar es la construcción de una red DHT extendida al conjunto de una infraestructura *grid*, de manera que la compartición de ficheros estuviera generalizada y la elección de una fuente u otra fuera una simple cuestión de preferencia. Así, los datos requeridos se

buscarían primero en los *pilots* del nodo local. De no encontrarse allí, se contactaría a los otros *pilots* del mismo *site*. Si esta tentativa tampoco tuviera éxito, se recurriría al SE local. Finalmente, si fuera necesario, se buscaría el fichero en *pilots* o SEs remotos. A nuestro entender, este es un modelo más elegante y homogéneo que el actual. Cabe indicar, sin embargo, que el procedimiento de asignación cooperativa de tareas seguiría, en principio, confinado a los límites de un *site* (y su red local).

Finalmente, y en términos más generales, aspiramos a profundizar en el conocimiento de cómo la microplanificación y la comunicación entre *pilots* puede contribuir a mejorar la ejecución de trabajos masivos en grandes infraestructuras de computación distribuidas.

List of Acronyms

ACK ..... Acknowledgment

ALICE..... A Large Ion Collider Experiment

AliEn..... ALICE Environment

AM ..... Application Master

API ..... Application Programming Interface

ATLAS ..... A Toroidal LHC Apparatus

BB ..... Bucket-based

CDN ..... Content Delivery Network

CE ..... Computing Element

CERN ..... European Laboratory of Particle Physics

CIEMAT ..... Centro de Investigaciones Energéticas Medioambientales y Tecnológicas

CMS ..... Compact Muon Solenoid

CPU ..... Central Processing Unit

CRAB ..... CMS Remote Analysis Builder

DB ..... Database

DBS ..... Dataset Bookkeeping Service

DHT ..... Distributed Hash Table

DIANA ..... Data Intensive And Network Aware

DIRAC ..... Distributed Infrastructure with Remote Agent Control

DLI ..... Data Location Interface

DLS ..... Data Location Service

DNS ..... Domain Name System

DPS ..... Data Placement System

EGI ..... European Grid Infrastructure

EMI ..... European Middleware Initiative

ERR ..... Empty Regions Re-assignment

F1 ..... Forward one

FA ..... Forward all

FIFO ..... First in, First out

GPU ..... Graphics Processing Unit

HDFS ..... Hadoop File System

HPC ..... High Performance Computing

HTC ..... High Throughput Computing

I/O ..... Input/Output

ID ..... Identifier

JSON ..... JavaScript Object Notation

LAN ..... Local Area Network

LFC ..... LCG File Catalog

LFN ..... Logical File Name

LHC ..... Large Hadron Collider

LHCb ..... Large Hadron Collider beauty

LP ..... Linear Programming

LRU ..... Least Recently Used

NP ..... Nondeterministic Polynomial time

OGSA ..... Open Grid Services Architecture

OSG ..... Open Science Grid

P2P ..... Peer-to-Peer

PA ..... ProdAgent

PanDA ..... Production and Distributed Analysis

PB ..... Partition-based

R1 ..... Replicate one

R1-F1 ..... Replicate one/Forward one

R1-FA ..... Replicate one/Forward all

RA ..... Replicate all

RA-F1 ..... Replicate all/Forward one

RA-FA ..... Replicate all/Forward all

REST ..... REpresentational State Transfer

SE ..... Storage Element

SPOF ..... Single Point of Failure

SQL ..... Structured Query Language

SRM ..... Storage Resource Manager

TCP ..... Transmission Control Protocol

TFC ..... Trivial File Catalog

TQ ..... Task Queue

TTL ..... Time-to-live

UI ..... User Interface

URL ..... Uniform Resource Locator

VM ..... Virtual Machine

VO ..... Virtual Organization

WAN ..... Wide Area Network

WLCG ..... Woldwide LHC Computing Grid

WMS ..... Workload Management System

WN ..... Worker Node

XOR ..... Exclusive Or

YARN ..... Yet Another Resource Negotiator

Bibliography

[1] G. Juve and E. Deelman. *Scientific Workflows and Clouds*. Crossroads, vol. 16(3), pp. 14–18, ACM, 2010.

[2] R. Bryant, R. H. Katz and E. D. Lazowska. *Big-Data Computing: Creating Revolutionary Breakthroughs in Commerce, Science and Society*, 2008.

[3] I. J. Taylor, E. Deelman *et al.* Workflows for e-Science. Springer-Verlag London Limited, 2007.

[4] I. Foster and C. Kesselman, eds. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[5] MIT. *10 Emerging Technologies That Will Change the World*, February 2003. Available at `http://www2.technologyreview.com/Infotech/13060/` (accessed: January 14th, 2015).

[6] R. Buyya, C. S. Yeo *et al. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility*. Future Generation computer systems, vol. 25(6), pp. 599–616, Elsevier, 2009.

[7] D. Parkhill. The Challenge of the Computer Utility. Addison-Wesley Educational Publishers Inc., 1966.

[8] I. Foster, C. Kesselman and S. Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. Intl. Journal of High Performance Computing Applications, vol. 15(3), pp. 200–222, Sage Publications, 2001.

[9] I. Foster, C. Kesselman and S. Tuecke. *What is the Grid? A Three Point Checklist*, July 2002.

[10] E. Huedo, R. S. Montero and I. M. Llorente. *A Framework for Adaptive Execution in Grids*. Software: Practice and Experience, vol. 34, pp. 631–651, Wiley Online Library, 2004.

[11] I. FOSTER and C. KESSELMAN. *Globus: A Metacomputing Infrastructure Toolkit.* Intl. Journal of High Performance Computing Applications, vol. 11(2), pp. 115–128, SAGE Publications, 1997.

[12] I. FOSTER, C. KESSELMAN *et al. The Physiology of the Grid.* Grid computing: making the global infrastructure a reality, pp. 217–249, John Wiley & Sons, 2003.

[13] I. FOSTER. *Globus Online: Accelerating and Democratizing Science through Cloud-based Services.* Internet Computing, IEEE, vol. 15(3), pp. 70–73, May 2011.

[14] EGI. *European Grid Infrastructure.* Available at `http://www.egi.eu/` (accessed: January 22th, 2015).

[15] OSG. *Open Science Grid.* Available at `http://www.opensciencegrid.org/` (accessed: January 22nd, 2015).

[16] I. BIRD. *Computing for the Large Hadron Collider.* Annual Review of Nuclear and Particle Science, vol. 61, pp. 99–118, 2011.

[17] EMI. *European Middleware Initiative.* Available at `http://www.eu-emi.eu` (accessed: January 22nd, 2015).

[18] I. BIRD, F. CARMINATI *et al. Update of the Computing Models of the WLCG and the LHC Experiments.* Tech. Rep. CERN-LHCC-2014-014. LCG-TDR-002, CERN, Geneva, April 2014.

[19] D. THAIN, T. TANNENBAUM and M. LIVNY. *Distributed Computing in Practice: The Condor Experience.* Concurrency and Computation: Practice and Experience, vol. 17(2-4), pp. 323–356, Wiley Online Library, 2005.

[20] A. DORIGO, P. ELMER *et al. Xrootd-A Highly Scalable Architecture for Data Access.* WSEAS Transactions on Computers, vol. 1(4.3), 2005.

[21] CERN. *Large Hadron Collider.* Available at `http://public.web.cern.ch/public/en/LHC/LHC-en.html` (accessed: January 5th, 2015).

[22] WLCG. *Worldwide LHC Computing Grid*, 2015. Available at `http://wlcg-public.web.cern.ch/` (accessed: January 19th, 2015).

[23] F. DONNO, L. ABADIE *et al. Storage Resource Manager Version 2.2: Design, Implementation, and Testing Experience.* In J. Phys.: Conf. Ser., vol. 119, p. 062028. IOP Publishing, 2008.

[24] P. ANDREETTO, S. ANDREOZZI *et al. The gLite Workload Management System.* In J. Phys.: Conf. Ser., vol. 119, p. 062007. IOP Publishing, 2008.

[25] CMS. *Compact Muon Solenoid.* Available at `http://cms.web.cern.ch/` (accessed: January 11th, 2015).

[26] J. HERNÁNDEZ, P. KREUZER *et al. CMS Monte Carlo Production in the WLCG Computing Grid.* In J. of Phys.: Conf. Series, vol. 119, p. 052019. IOP Publishing, 2008.

[27] E. FAJARDO, O. GUTSCHE *et al. A New Era for Central Processing and Production in CMS.* In J. of Phys.: Conf. Series, vol. 396, p. 042018. IOP Publishing, 2012.

[28] D. SPIGA, S. LACAPRARA *et al. The CMS Remote Analysis Builder (CRAB).* In High Performance Computing–HiPC 2007, pp. 580–586. Springer, 2007.

[29] I. SFILIGOI. *glideinWMS–a Generic Pilot-based Workload Management System.* In J. Phys.: Conf. Ser., vol. 119, p. 062044. IOP Publishing, 2008.

[30] J. REHN, T. BARRASS *et al. PhEDEx High-throughput Data Transfer Management System.* Computing in High Energy and Nuclear Physics (CHEP) 2006, 2006.

[31] A. DELGADO PERIS, A. FANFANI *et al. Data Location, Transfer and Bookkeeping in CMS.* Nuclear Phys. B-Proceedings Supplements, vol. 177, pp. 279–280, Elsevier, 2008.

[32] P. MELL and T. GRANCE. *The NIST Definition of Cloud Computing.* National Institute of Standards and Technology, 2011.

[33] AMAZON. *Amazon Web Services.* Available at `http://aws.amazon.com` (accessed: January 17th, 2015).

[34] I. FOSTER, Y. ZHAO *et al. Cloud Computing and Grid Computing 360-Degree Compared.* In Grid Computing Environments Workshop, 2008. GCE'08, pp. 1–10. Ieee, 2008.

[35] S. OSTERMANN, A. IOSUP *et al. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing.* In Cloud Computing, pp. 115–131. Springer, 2010.

[36] D. EVANS, I. FISK *et al. Using Amazon's Elastic Compute Cloud to Dynamically Scale CMS Computational Resources.* In J. of Phys.: Conf. Series, vol. 331, p. 062031. IOP Publishing, 2011.

[37] R. MEDRANO LLAMAS, M. CINQUILLI *et al.* *Commissioning the CERN IT Agile Infrastructure with Experiment Workloads.* In J. of Phys.: Conf. Series, vol. 513, p. 032066. IOP Publishing, 2014.

[38] P. RUSSOM *et al.* *Big Data Analytics.* TDWI Best Practices Report, Fourth Quarter, TDWI Research, 2011.

[39] GARTNER. *Press release: Gartner's 2014 Hype Cycle for Emerging Technologies Maps the Journey to Digital Business*, August 2014. Available at `http://www.gartner.com/newsroom/id/2819918` (accessed: January 19th, 2015).

[40] B. BOCKELMAN. *Using Hadoop as a Grid Storage Element.* In Journal of physics: Conf. series, vol. 180, p. 012047. IOP Publishing, 2009.

[41] K. RANGANATHAN and I. FOSTER. *Decoupling Computation and Data Scheduling in Distributed Data-intensive Applications.* In High Performance Distributed Computing, 2002. HPDC-11 2002. Proc. 11th IEEE Intl. Symp. on, pp. 352–358. IEEE, 2002.

[42] J. DEAN and S. GHEMAWAT. *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, vol. 51(1), pp. 107–113, ACM, 2008.

[43] T. AGERWALA. *A New Paradigm for Computing: Data-Centric Systems.* Wired, December 2014. Available at `http://www.wired.com/2014/12/data-centric-systems/` (accessed: January 20th, 2015).

[44] A. CHERVENAK, I. FOSTER *et al.* *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets.* Journal of network and computer applications, vol. 23(3), pp. 187–200, Elsevier, 2000.

[45] P. J. DENNING. *The Working Set Model for Program Behavior.* Communications of the ACM, vol. 11(5), pp. 323–333, ACM, 1968.

[46] M. DALHEIMER, F.-J. PFREUNDT and P. MERZ. *Agent-based Grid Scheduling with Calana.* In Parallel Processing and Applied Mathematics, pp. 741–750. Springer, 2006.

[47] M. D. DE ASSUNCAO and R. BUYYA. *An Evaluation of Communication Demand of Auction Protocols in Grid Environments.* In Proc. of the 3rd Intl. Workshop on Grid Economics & Business (GECON 2006), vol. 16, 2006.

[48] ATLAS. *ATLAS@HOME Project.* Available at `http://atlasathome.cern.ch` (accessed: February 17th, 2015).

[49] C. PINCHAK, P. LU and M. GOLDENBERG. *Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences.* In Job Scheduling Strategies for Parallel Processing, pp. 205–228. Springer, 2002.

[50] J. BERTHOLD, M. DIETERLE *et al. Hierarchical Master-Worker Skeletons.* In Practical Aspects of Declarative Languages, pp. 248–264. Springer, 2008.

[51] A. TSAREGORODTSEV, V. GARONNE *et al. DIRAC–Distributed Infrastructure with Remote Agent Control.* In Proc. of CHEP2003, 2003.

[52] S. BAGNASCO, L. BETEV *et al. AliEn: ALICE Environment on the Grid.* In J. Phys.: Conf. Ser., vol. 119, p. 062012. IOP Publishing, 2008.

[53] T. MAENO. *PanDA: Distributed Production and Distributed Analysis System for ATLAS.* In J. Phys.: Conf. Ser., vol. 119, p. 062036. IOP Publishing, 2008.

[54] D. GROEP, O. KOEROO and G. VENEKAMP. *gLExec: Gluing Grid Computing to the Unix World.* In J. Phys.: Conf. Ser., vol. 119, p. 062032. IOP Publishing, 2008.

[55] G. A. STEWART, D. CAMERON *et al. Storage and Data Management in EGEE.* In Proc. of the fifth Australasian symposium on ACSW frontiers-Volume 68, pp. 69–77. Australian Computer Society, Inc., 2007.

[56] P. CANAL, B. BOCKELMAN and R. BRUN. *ROOT I/O: The Fast and Furious.* In J. Phys.: Conf. Ser., vol. 331, p. 042005. IOP Publishing, 2011.

[57] M. BENCIVENNI, F. BONIFAZI *et al. A Comparison of Data-Access Platforms for the Computing of Large Hadron Collider Experiments.* Nuclear Science, IEEE Transactions on, vol. 55(3), pp. 1621–1630, IEEE, 2008.

[58] A. J. PETERS and L. JANYST. *Exabyte Scale Storage at CERN.* In J. Phys.: Conf. Ser., vol. 331, p. 052015. IOP Publishing, 2011.

[59] G. L. PRESTI, O. BARRING *et al. CASTOR: A Distributed Storage Resource Facility for High Performance Data Processing at CERN.* In MSST, vol. 7, pp. 275–280. Citeseer, 2007.

[60] D. G. CAMERON, R. CARVAJAL-SCHIAFFINO *et al. Evaluating Scheduling and Replica Optimisation Strategies in OptorSim.* In Proc. of the 4th Intl. Workshop on Grid Computing, p. 52. IEEE Computer Society, 2003.

[61] R. McClatchey, A. Anjum *et al. Data Intensive and Network Aware (DIANA) Grid Scheduling.* Journal of Grid Computing, vol. 5(1), pp. 43–64, Springer, 2007.

[62] S. Venugopal, R. Buyya and L. Winton. *A Grid Service Broker for Scheduling Distributed Data-oriented Applications on Global Grids.* In Proc. of the 2nd workshop on Middleware for grid computing, pp. 75–80. ACM, 2004.

[63] T. Maeno, K. De *et al. Evolution of the ATLAS PanDA Production and Distributed Analysis System.* In J. Phys.: Conf. Ser., vol. 396, p. 032071. IOP Publishing, 2012.

[64] A. Chervenak, E. Deelman *et al. Data Placement for Scientific Applications in Distributed Environments.* In Proc. of the 8th IEEE/ACM Intl. Conf. on Grid Computing, pp. 267–274. IEEE Computer Society, 2007.

[65] M. Korupolu, A. Singh and B. Bamba. *Coupled Placement in Modern Data Centers.* In Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE Intl. Symp. on, pp. 1–12. IEEE, 2009.

[66] Apache. *The Hadoop Project.* Available at `http://hadoop.apache.org/` (accessed: February 7th, 2015).

[67] S. Ghemawat, H. Gobioff and S.-T. Leung. *The Google File System.* In ACM SIGOPS operating systems review, vol. 37, pp. 29–43. ACM, 2003.

[68] K. Shvachko, H. Kuang *et al. The Hadoop Distributed File System.* In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symp. on, pp. 1–10. IEEE, 2010.

[69] J. Bent, D. Rotem *et al. Coordination of Data Movement with Computation Scheduling on a Cluster.* In Challenges of Large Applications in Distributed Environments, 2005. CLADE 2005. Proc., pp. 25–34. IEEE, 2005.

[70] S. Shankar and D. J. DeWitt. *Data Driven Workflow Planning in Cluster Management Systems.* In Proc. of the 16th Intl. Symp. on High Performance Distributed Computing, HPDC '07, pp. 127–136. ACM, New York, NY, USA, 2007.

[71] V. K. Vavilapalli, A. C. Murthy *et al. Apache Hadoop YARN: Yet Another Resource Negotiator.* In Proc. of the 4th annual Symp. on Cloud Computing, p. 5. ACM, 2013.

[72] M. ZAHARIA, M. CHOWDHURY *et al. Spark: Cluster Computing with Working Sets.* In Proc. of the 2nd USENIX conf. on Hot Topics in Cloud Computing, pp. 10–10, 2010.

[73] M. ZAHARIA, D. BORTHAKUR *et al. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling.* In Proc. of the 5th European Conf. on Computer systems, pp. 265–278. ACM, 2010.

[74] M. SCHWARZKOPF, A. KONWINSKI *et al. Omega: Flexible, Scalable Schedulers for Large Compute Clusters.* In Proc. of the 8th ACM European Conf. on Computer Systems, pp. 351–364. ACM, 2013.

[75] V. GARONNE, G. A. STEWART *et al. The ATLAS Distributed Data Management Project: Past and Future.* In J. Phys.: Conf. Ser., vol. 396, p. 032045. IOP Publishing, 2012.

[76] K. BLOOM, C. COLLABORATION *et al. CMS Use of a Data Federation.* In J. Phys.: Conf. Ser., vol. 513, p. 042005. IOP Publishing, 2014.

[77] H. BALAKRISHNAN, M. KAASHOEK *et al. Looking up Data in P2P Systems.* Communications of the ACM, vol. 46(2), pp. 43–48, ACM, 2003.

[78] M. RIPEANU. *Peer-to-Peer Architecture Case Study: Gnutella Network.* In Peer-to-Peer Computing, 2001. Proceedings. First International Conference on, pp. 99–100. IEEE, 2001.

[79] P. MAYMOUNKOV and D. MAZIERES. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.* In Revised Papers from the First Intl. Workshop on Peer-to-Peer Systems (IPTPS '01), pp. 53–65. Springer-Verlag, London, UK, 2002.

[80] D. KARGER, E. LEHMAN *et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.* In Proc. of the 29th Annual ACM Symp. on Theory of Computing, STOC '97, pp. 654–663. ACM, New York, NY, USA, 1997.

[81] S. EL-ANSARY and S. HARIDI. *An Overview of Structured P2P Overlay Networks.* Swedish Institute of Computer Science and Royal Institute of Technology, 2004.

[82] H. ZHANG, Y. WEN *et al. DHT Applications.* In Distributed Hash Table, SpringerBriefs in Computer Science, pp. 39–55. Springer New York, 2013.

[83] P. DRUSCHEL and A. ROWSTRON. *PAST: A Large-scale, Persistent Peer-to-Peer Storage Utility.* In Hot Topics in Operating Systems, 2001. Proc. of the Eighth Workshop on, pp. 75–80. IEEE, 2001.

[84] I. CLARKE, O. SANDBERG *et al. Freenet: A Distributed Anonymous Information Storage and Retrieval System.* In Designing Privacy Enhancing Technologies, pp. 46–66. Springer, 2001.

[85] B. FITZPATRICK. *Distributed Caching with Memcached.* Linux journal, vol. 2004(124), p. 5, Belltown Media, 2004.

[86] A. CHAZAPIS, A. ZISSIMOS and N. KOZIRIS. *A Peer-to-Peer Replica Management Service for High-Throughput Grids.* In Intl. Conf. on Parallel Processing, 2005, pp. 443–451. IEEE, 2005.

[87] Y. YANG, K. LIU *et al. Peer-to-Peer Based Grid Workflow Runtime Environment of SwinDeW-G.* In IEEE Intl. Conf. on e-Science and Grid Computing, pp. 51–58. IEEE, 2007.

[88] J. CAO, O. M. KWONG *et al. A Peer-to-Peer Approach to Task Scheduling in Computation Grid.* In Grid and Cooperative Computing, pp. 316–323. Springer, 2004.

[89] M. RAHMAN, R. RANJAN and R. BUYYA. *Cooperative and Decentralized Workflow Scheduling in Global Grids.* Future Generation Computer Systems, vol. 26(5), pp. 753–768, Elsevier, 2010.

[90] I. STOICA, R. MORRIS *et al. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.* ACM SIGCOMM Computer Communication Review, vol. 31(4), pp. 149–160, ACM, 2001.

[91] A. ROWSTRON and P. DRUSCHEL. *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.* In Middleware 2001, pp. 329–350. Springer, 2001.

[92] M. STEINER, T. EN-NAJJARY and E. W. BIERSACK. *A Global View of Kad.* In Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, pp. 117–122. ACM, 2007.

[93] J. POUWELSE, P. GARBACKI *et al. The Bittorrent P2P File-sharing System: Measurements and Analysis.* In Peer-to-Peer Systems IV, pp. 205–216. Springer, 2005.

[94] J. LI, K. SOLLINS and D. LIM. *Implementing Aggregation and Broadcast over Distributed Hash Tables.* ACM SIGCOMM Computer Communication Review, vol. 35(1), pp. 81–92, ACM, 2005.

[95] A. GHODSI, L. ONANA ALIMA *et al. Self-Correcting Broadcast in Distributed Hash Tables.* In 15th IASTED Intl. Conf., Parallel and Distributed Computing and Systems (PDCS). ACTA Press, 2003.

[96] F. LIN, C. HENRICSSON *et al. HyperCircle: An Efficient Broadcast Protocol for Super-Peer P2P Networks.* In Intl. Conf. on Computational Science and Engineering (CSE), 2009, vol. 2, pp. 426–433. IEEE, 2009.

[97] S. EL-ANSARY, L. ALIMA *et al. Efficient Broadcast in Structured P2P Networks.* In 2nd Intl. Workshop On Peer-To-Peer Systems (IPTPS'03), pp. 304–314. Springer, 2003.

[98] K. HUANG and D. ZHANG. *DHT-based Lightweight Broadcast Algorithms in Large-Scale Computing Infrastructures.* Future Generation Computer Systems, vol. 26(3), pp. 291–303, Elsevier, 2010.

[99] W. LI, S. CHEN *et al. An Efficient Broadcast Algorithm in Distributed Hash Table under Churn.* In Intl. Conf. on Wireless Communications, Networking and Mobile Computing (WiCom), 2007, pp. 1929–1932. IEEE, 2007.

[100] M. WAHLISCH, T. SCHMIDT and G. WITTENBURG. *Broadcasting in Prefix Space: P2P Data Dissemination with Predictable Performance.* In Fourth Intl. Conf. on Internet and Web Applications and Services (ICIW'09), pp. 74–83. IEEE, 2009.

[101] A. DELGADO PERIS, J. M. HERNÁNDEZ and E. HUEDO. *Evaluation of the Broadcast Operation in Kademlia.* In IEEE 14th Intl. Conf. on High Performance Computing and Communication & IEEE 9th Intl. Conf. on Embedded Software and Systems (HPCC-ICESS), pp. 756–763. IEEE Computer Society, 2012.

[102] Z. CZIRKOS and G. HOSSZÚ. *Solution for the Broadcasting in the Kademlia Peer-to-Peer Overlay.* Computer Networks, vol. 57(8), pp. 1853–1862, Elsevier, 2013.

[103] P. MERZ and K. GORUNOVA. *Efficient Broadcast in P2P Grids.* In IEEE Intl. Symp. on Cluster Computing and the Grid (CCGrid), 2005, vol. 1, pp. 237–242. IEEE, 2005.

[104] J. MOSCICKI, M. LAMANNA *et al. Processing Moldable Tasks on the Grid: Late Job Binding with Lightweight User-level Overlay.* Future Generation Computer Systems, vol. 27(6), pp. 725 – 736, 2011.

[105] J. HERNÁNDEZ, D. EVANS and S. FOULKES. *Multi-core Processing and Scheduling Performance in CMS.* In J. Phys.: Conf. Ser., vol. 396, p. 032055. IOP Publishing, 2012.

[106] H. STOCKINGER, F. DONNO *et al. Matchmaking, Datasets and Physics Analysis.* In Parallel Processing, 2005. ICPP 2005 Workshops. Intl. Conf. Workshops on, pp. 21–28. IEEE, 2005.

[107] J.-P. BAUD, J. CASEY *et al. Performance Analysis of a File Catalog for the LHC Computing Grid.* In High Performance Distributed Computing, 2005. HPDC-14. Proc. 14th IEEE Intl. Symp. on, pp. 91–99. IEEE, 2005.

[108] D. CROCKFORD. *The Application/json Media Type for Javascript Object Notation (JSON)*, June 2006. RFC 4627.

[109] S. K. PATERSON and A. TSAREGORODTSEV. *DIRAC Optimized Workload Management.* In J. Phys.: Conf. Ser., vol. 119, p. 062040. IOP Publishing, 2008.

[110] D. BRADLEY, T. ST CLAIR *et al. An Update on the Scalability Limits of the Condor Batch System.* In J. Phys.: Conf. Ser., vol. 331, p. 062002. IOP Publishing, 2011.

[111] J. D. ULLMAN. *NP-Complete Scheduling Problems.* Journal of Computer and System Sciences, vol. 10(3), pp. 384–393, Elsevier, 1975.

[112] A. DELGADO PERIS, J. M. HERNÁNDEZ *et al. Data Location-aware Job Scheduling in the Grid. Application to the Gridway Metascheduler.* In Journal of Physics: Conference Series, vol. 219, p. 062043. IOP Publishing, 2010.

[113] A. DELGADO PERIS, J. M. HERNÁNDEZ and E. HUEDO. *Evaluation of Alternatives for the Broadcast Operation in Kademlia under Churn.* Peer-to-Peer Networking and Applications, Springer, 2015. DOI: `http://dx.doi.org/10.1007/s12083-015-0338-y`.

[114] K. HASHAM, A. DELGADO PERIS *et al. CMS Workflow Execution Using Intelligent Job Scheduling and Data Access Strategies.* IEEE Transactions on Nuclear Science, vol. 58(3), pp. 1221–1232, IEEE, 2011.

[115] A. DELGADO PERIS, J. M. HERNÁNDEZ and E. HUEDO. *Distributed Scheduling and Data Sharing in Late-binding Overlays.* In High Performance Computing Simulation (HPCS), 2014 Intl. Conf. on, pp. 129–136, July 2014.

[116] CHERRYPY. *A Minimalist Python Web Framework.* Available at `http://cherrypy.org/` (accessed: April 9th, 2015).

[117] ASYNCORE. *Asynchronous Socket Handler.* Available at `http://docs.python.org/2/library/asyncore.html` (accessed: April 13th, 2015).